Rec'd PCT/PTC 03 MAY 2005 #2
PC AU03/01473

REC'D 01 DEC 2003
WIPO PCT

**Patent Office**
**Canberra**

I, JANENE PEISKER, TEAM LEADER EXAMINATION SUPPORT AND SALES hereby certify that annexed is a true copy of the Provisional specification in connection with Application No. 2003901926 for a patent by CODE VALLEY PTY LIMITED as filed on 22 April 2003.

WITNESS my hand this
Twentieth day of November 2003

J.K.H.

JANENE PEISKER
TEAM LEADER EXAMINATION
SUPPORT AND SALES

COMMONWEALTH OF AUSTRALIA
AUSTRALIA
PATENT OFFICE

**PRIORITY DOCUMENT**
SUBMITTED OR TRANSMITTED IN COMPLIANCE WITH RULE 17.1(a) OR (b)

BEST AVAILABLE COPY

Our Ref: 7762410

P/00/009
Regulation 3:2

AUSTRALIA

Patents Act 1990

# PROVISIONAL SPECIFICATION

Applicant(s): Code Valley Pty Limited
Suite 3, Floor 4
75 Denham Street
Townsville Queensland 4810
Australia

Address for Service: DAVIES COLLISON CAVE
Patent & Trade Mark Attorneys
Level 10, 10 Barrack Street
SYDNEY NSW 2000

Invention Title: **Code generation**

The invention is described in the following statement:

# CODE GENERATION

## Background of the Invention

The present invention relates to a method and apparatus for generating computer executable code, and in particular, to generating computer executable code using components, each of which corresponds to a respective service for manipulating data.

## Description of the Prior Art

The reference to any prior art in this specification is not, and should not be taken as, an acknowledgement or any form of suggestion that the prior art forms part of the common general knowledge in Australia.

It is clear that software is developed in a competitive environment but history has detailed a flat productivity curve over the last thirty years. Any gains are insignificant compared to what has been achieved in other industries over the same period. This disparity is unique to the software industry and is a direct result of how software is constructed.

The majority of computer software is constructed through a manual process, utilising programmers to generate code for respective applications software projects. Each software application will generally be created using one or more programmers, to create the software application on a case-by-case basis, with little or no code reuse.

One of the reasons behind this is that typically only a limited number of entities (typically one company), which will invest in the development of any one software application. As a result, entities are generally unwilling to invest more than necessary in the development of software code. This means that once a functioning application is developed, little money and effort is invested in optimising the code forming the application.

Other reasons include:

- Initially favouring an in-house model of development over a more difficult distributed model, which thereby required increased generalisation of limited resources.
- Introduction of the Library/Linker, which established standard routines for performing predetermined functions, thereby reducing competitiveness and optimisation; and
- The prevailing view of the very nature of software production, which requires customers to accept limits imposed on them by the programmers.

As a result of this, programmers have to be generalists that are capable of programming software to perform a wide range of functionality, allowing them to compete in the market place as it stands. Thus, it will be appreciated that in the current software programming environment, there is little call for a programmer that is very good at only one very minor programming task, when generalists are available that can adequately code entire programs.

This lack of specialisation leads to a number of problems in the field of software creation, including for example:

- Bugs or mistakes;
- Software bloating;
- Complexity limits; and,
- Low barrier to entry.

Bugs and mistakes arise, to a large extent, due to the large amounts of code each programmer must write. This has a number of problems such as limiting the amount of time the programmer can physically spend writing the code and increasing the number of variables the programmer must consider. Bugs are an extensive problem within the current software programming techniques. The result of this is additional time has been

spent in correcting bugs and mistakes that arise, through the use of software patches, or upgrades, as well as correcting the after effects, such as damage caused by viruses, or the like.

Software bloating is another effect associated with the lack of specialisation within the programming field. As with any generalist, refining a product comes at the expense of productivity. In particular, a programmer may be able to construct a functioning program relatively quickly. However, optimising the code to minimise the code quantity, whilst improving operation can take a long time for only minimal improvements. The programmer's skill at optimising would generally also be rudimentary given the individual's knowledge is spread over many fields and similarly the coder gets paid by getting the product on the market. The result is a requirement for more powerful machines to handle the unnecessary size and complexity of modern software.

The complexity limit arises due to the fact that as the level of complexity rises, it becomes impossible for one person to understand all aspects of a particular software development effort. Most industries that have become highly specialised can field large complex projects whereas software development has not reached, let alone attained, any degree of specialisation.

In general, any industry has a barrier to entry that is proportional to its maturity. Well-developed industries like semiconductors have incredible barriers that even nations balk at tackling. A feature of industrialisation is the large amounts of capital that are required to set up a business. However, in the case of more basic industries, such as craft industries, it is possible for anyone to enter the industry with dedication and a modest outlay, and achieve best practice. This is an indication that the software industry is undeveloped, as it has a minimal barrier to entry.

The effects of these problems are endemic within the software field. Studies have shown that "for every six, new, large-scale software systems that are put into operation, two others are cancelled. Indeed so severe is this software crisis that three quarters of large-scale systems commissioned are operating failures", either they do not function as intended or they are not used at all.

A committee consisting of over 50 top programmers, computer scientists and industry leaders first addressed this problem at the 1968 NATO Science Meeting. This Committee was given the task of finding a way out of the "software crisis".

A number of attempts have been made to solve the problems, including:

- Development of 3rd, 4th and 5th Generation Languages, which seek to abstract the programmer away from machine code, are responsible for some of the early successes in productivity. So successful was the early productivity increases that languages continue today as the main thrust in the quest for productivity improvement.
- Object Oriented Programming, a new type of abstraction encapsulating data with code used to process that data, is achieving limited success mainly through controlling complexity.
- Computer Aided Software Engineering (CASE), which seeks to assist in managing the complexity of large software development.
- Code Reuse, which is an effort to reuse previous intellectual endeavour.
- Formal Methods, which use mathematical proofs to verify correctness and to acutely address the large numbers of defects and bugs synonymous with software development.
- Decompilers that extract intellectual content from historical code.

Despite these initiatives however, little headway is being made in improving software production.

In particular, there remains little specialisation within the industry, with the majority of software applications being generated on a case by case basis, by a limited number of programmers. Accordingly, methods like 3rd, 4th, 5th GLs, OOP, CASE, Formal Methods, Decompilers and countless others do not address the problem but only the symptoms.

It can therefore be seen that thirty years after the NATO conference only minimal progress if any has been made, and "the vast majority of computer code is still hand-crafted from raw programming languages by artisans using techniques they neither measure nor are able to repeat consistently".

**Summary of the Present Invention**

In a first broad form the present invention provides a method of generating computer executable code using components, each component corresponding to a respective service for manipulating data in a predetermined manner, the method including:

a) Combining a number of components such that the combined components define a sequence of data manipulations; and,
b) Implementing the combined components such that the defined sequence of data manipulations is performed, to thereby generate the computer executable code.

Usually at least some of the components include:

a) One or more inputs for receiving data to be manipulated; and,
b) One or more outputs for outputting manipulated data, the method including combining the components by interconnecting selected ones of the outputs and inputs of respective components.

Typically:

a) Each input is associated with an input specification, each input specification indicating one or more data types representing types of data the input is adapted to receive; and,

b) Each output is associated with an output specification, each output specification indicating one or more data types representing the types of data adapted to be provided at the respective output.

The method of connecting a first component to a second component typically includes:

a) Selecting at least one output of the first component;
b) Selecting at least one input of the second component;
c) Comparing the input and output specifications of the selected components; and,
d) Interconnecting the input and the output in response to a successful comparison.

Each input and each output is usually associated with a respective agent, the method including causing the agents to cooperate to compare the input and output specifications.

The method can include causing the agents to successively compare the data types of each specification until a successful comparison is performed.

Each agent is generally formed from computer executable code, the agents being adapted to be implemented by a suitably programmed processing system.

The data received at the input can include a sequence of one or more data portions, the manipulations including:

a) Adding data portions into the sequence at a predetermined location;

b) Moving data portions from a first location to a second location within the sequence; and,

c) Removing data portions from the sequence.

Typically, at least a portion of the method is performed using a processing system including a store, the method manipulations including storing one or more of the data portions in the store.

Usually, at least some of the components are formed from a number of combined sub-components, the sub-components being components.

Preferably, one or more of the services are performed using at least one of:

a) Manual manipulation of the data by an individual;

b) Computer executable code adapted to be executed by a processing system, to thereby manipulate of the data automatically; and,

c) A component schematic.

The method can be performed using one or more processing systems.

The method can include causing a first processing system to:

a) Select a number of components in response to input commands received from a user;

b) Combine selected components; and,

c) Cause the combined components to be implemented such that the defined sequence of data manipulations is performed.

Typically the method includes causing the first processing system to:

a) Generate a graphical representation of the selected components; and,

b) Manipulate the graphical representation in response to input commands received from a user to thereby define the combined components.

The components preferably include one or more inputs for receiving data to be manipulated, and one or more outputs for outputting manipulated data, the method including causing the processor to:

a) Provide an indication of the inputs and outputs of each selected component to the user; and,
b) Interconnect selected ones of the outputs and inputs in response to input commands from the user.

The method may include using a second processing system, the second processing system being adapted to:

a) Determine a number of components;
b) Provide an indication of the components to the user via the first processing system; and,
c) Select respective ones of the components in response to input commands from the user.

The second processing system can include:

a) A store for storing at component specifications, each component specification indicating the service provided by a respective component; and,
b) A processor adapted to:
    i) Access the component specifications stored in the store; and,
    ii) Provide an indication of the services provided by the components to the user via the first processing system.

The store can be further adapted to store:

a) Input specifications, each input specification indicating one or more data types representing types of data a respective input is adapted to receive; and,

b) Output specifications, each output specification indicating one or more data types representing the types of data adapted to be provided at a respective output; and,

The processor being adapted to:

i) Access the input and/or output specifications stored in the store; and,

ii) Provide an indication of the data types to the user, thereby allowing the user to select the components.

The first and second processing systems can be coupled together via a communications network.

The method may include causing the second processing system to:

a) Receive a component request from the first processing system ;

b) Transfer an indication of one or more components to the first processing system in accordance with the request, the indication including an indication of at least the service provided by each respective component; and,

c) Select one or more components in accordance with a component selection received from the first processing system.

The method may further include causing the second processing system to:

a) Obtain a graphical representation of the selected components;

b) Transfer the graphical representations to the first processing system.

The components generally including one or more inputs for receiving data to be manipulated, and one or more outputs for outputting manipulated data. In this case, the method can include causing the first processing system to:

a) Allow the user to define a graphical interconnection between inputs and the outputs of the selected components; and,
b) Determine if the interconnections are acceptable; and,
c) Interconnect the components in response to a successful determination.

The first or second processing system may be adapted to store:

a) Input specifications, each input specification indicating one or more data types representing types of data a respective input is adapted to receive; and,
b) Output specifications, each output specification indicating one or more data types representing the types of data adapted to be provided at a respective output; and,

The first or second processing system being adapted to:

i) Access the input and/or output specifications stored in the store; and,
ii) Compare the input and output specifications of the selected components; and,
iii) Interconnect the input and the output in response to a successful comparison.

The method generally includes causing the first processing system to implementing the combined components in accordance with the generated graphical representation.

The store can be a database.

The first processing system can be coupled to one or more component processing systems via a communications network. In this case, each component processing system can be adapted to implement one or more of the components, the method including causing the first processing system to generate the computer executable code by:

a) Determining the next component to perform manipulation of the data;
b) Transferring the data to be manipulated to the respective component processing system via the communications network, the component processing system being adapted to perform the data manipulation;

c) Receiving the manipulated data from the component processing system; and,
d) Repeating steps (a) to (c) for each subsequent service.

The computer executable code can be generated by a generation entity, the generation entity being adapted to:

a) Provide a number of components, each component corresponding to a respective service for manipulating data in a predetermined manner; and,
b) Cause the software to be generated in accordance with a specified combination of the components, the specified combination defining a sequence of data manipulations that result in the generation of the desired executable code.

The components are typically provided by respective entities, the method including causing the generation entity to:

a) Determine the next component in the sequence;
b) Transfer the data to be manipulated to the respective entity;
c) Receive the manipulated data from the entity; and,
d) Repeat steps (a) to (c) to generate the computer executable code.

The method usually further includes causing the generation entity to:

a) Determine performance information, the performance information being representative of one or more criteria regarding the implementation of the components;
b) Provide the performance information to a user, the user selecting the components in accordance with the performance information.

The performance information can include at least one of:

a) An indication of the entity implementing the component;
b) An indication of the geographical location of the entity;

c) An indication of the duration for implementing the component;

d) An indication of a cost associated with implementing the respective component; and,

e) A rating, the rating being indicative of the success of the component.

The method generally includes:

a) Providing a number of different components for performing equivalent services, the different components being provided by different entities; and,

b) Inducing competition between the entities to thereby drive improvement of the components.

The method can also include generating revenue by charging a cost for the use of each component.

The method can be performed by a generation entity, in which case the method can include:

a) Providing at least some of the revenue to a respective entity implementing the component; and,

b) Having the software generation entity retain at least some of the revenue.

In a second broad form the present invention provides apparatus for generating computer executable code using components, each component corresponding to a respective service for manipulating data in a predetermined manner, the apparatus including one or more processing systems adapted to:

a) Combine a number of components such that the combined components define a sequence of data manipulations; and,

b) Implement the combined components such that the defined sequence of data manipulations is performed, to thereby generate the computer executable code.

The apparatus typically includes:

a) One or more component processing systems, each component processing system being adapted to implement a respective component; and,
b) A first processing system, the first processing system being adapted to:
    i) Define a combination of components in accordance with input commands received from a user; and,
    ii) Cause the number of components to be implemented such that the combined components define a sequence of data manipulations; and,

The first processing system can be adapted to:

a) Determine the components to be implemented;
b) Determine the component processing systems implementing the respective components; and
c) Transfer service requests to each of the determined component processing systems, the component processing systems being adapted to implement the respective service in response to the received service request.

The component processing system may be adapted to:

a) Receive the service request;
b) Generate a respective component instance; and,
c) Perform the service using the respective component instance.

The apparatus typically includes a second processing system, the second processing system being adapted to store details of available components.

The second processing system can be adapted to obtain the details of a component from a respective component processing system.

The first processing system may be adapted to cooperate with the second processing system to thereby allow a user to:

a) Select one or more of the available components; and
b) Define the component combination.

Typically the apparatus is adapted to perform the method of the first broad form of the invention.

In a third broad form the present invention provides a computer program product for generating computer executable code using components, each component corresponding to a respective service for manipulating data in a predetermined manner, the computer program product including computer executable code which when executed on a suitable processing system causes the processing system to perform the method of the first broad form of the invention.

In a fourth broad form the present invention provides a method of combining components using a processing system, each component representing a respective service for manipulating data, the method including:

a) Selecting one or more components to be combined; and,
b) Causing the processing system to:
    i) Generate a graphical representation of each selected component on a display; and,
    ii) Manipulate the graphical representation to define connections between inputs and outputs of the components.

The method typically includes causing the processing system to generate:

a) Component data representing the components in the representation.

b) Connection data representing the connections between the components.

The method typically includes:

a) Causing the processing system to generate a representation of a root component; and,
b) Defining connections between any inputs and outputs of the root component and inputs and outputs of the components.

The method generally includes causing the processing system to manipulate the graphical representation to display sub components associated with any one of the components

The method can include causing the processing system to manipulate the graphical representation in responds to input commands from the user.

The method may typically includes manipulating data in the method of the first broad form of the invention.

In a fifth broad form the present invention provides apparatus for combining components, each component representing a respective service for manipulating data, the apparatus including a processing system adapted to:

a) Select one or more components to be combined;
b) Generate a graphical representation of each selected component on a display; and,
c) Manipulate the graphical representation to define connections between inputs and outputs of the components.

The apparatus is generally adapted to perform the method of the fourth broad form of the invention.

In a sixth broad form the present invention provides a computer program product for combining components using a processing system, each component representing a respective service for manipulating data, the computer program product including computer executable code which when executed on a suitable processing system causes the processing system to perform the method of the fourth broad form of the invention.

In a seventh broad form the present invention provides a method of manipulating data by implementing one or more components, each component corresponding to a respective service for manipulating data, the method including:

a) Determining:
    i) The one or more components to be implemented;
    ii) Connections between inputs and outputs of respective ones of the components;
    iii) Any data to be manipulated;
b) For each component, requesting the provision of a respective service;
c) Transferring any data to be manipulated to one or more of the components, each entity being responsive to the data to perform the respective service and provide manipulated data at one or more outputs;
d) Transferring manipulated data between the outputs and inputs in accordance with the connections; and,
e) Obtaining manipulated data from one or more of the components.

The service associated with each component is typically performed by at least one of:

a) Manual manipulation of the data by an entity;
b) A processing system adapted to perform the respective service.

The method can include requesting the provision of a respective service by transferring a purchase order to each component, each component being responsive to the purchase order to provide a respective component instance.

The method generally includes:

a) Activating an agent associated with each input and each output of the respective components;
b) Causing each agent to cooperate with another agent in accordance with the determined connections; and,
c) Causing the agents to transfer the data between the components.

The agents can be adapted to:

a) Cooperate with other agents to establish a connection;
b) Receive data from an agent;
c) Transfer data to an agent;
d) Cooperate with other agents to perform agent hand-off.

Each agent is typically formed from executable code executed by the processing system.

The method of manipulating data may be used in the method of the first broad form of the invention.

In a eighth broad form the present invention provides apparatus for manipulating data by implementing one or more components, each component corresponding to a respective service for manipulating data, the apparatus including a processing system adapted to:

a) Determine:
    i) The one or more components to be implemented;

ii) Connections between inputs and outputs of respective ones of the components;

iii) Any data to be manipulated;

b) For each component, request the provision of a respective service;

c) Transfer any data to be manipulated to one or more of the components, each entity being responsive to the data to perform the respective service and provide manipulated data at one or more outputs;

d) Transferring manipulated data between the outputs and inputs in accordance with the connections; and,

e) Obtaining manipulated data from one or more of the components.

Each component can be implemented by a respective component processing system, in which case the processing system is adapted to request the provision of a respective service by transferring a request to the respective component processing system.

The apparatus can be adapted to perform the method of the seventh broad form of the invention.

In a ninth broad form the present invention provides a computer program product for manipulating data by implementing one or more components, each component corresponding to a respective service for manipulating data, the computer program product including computer executable code which when executed on a suitable processing system causes the processing system to perform the method of the seventh broad form of the invention.

In a tenth broad form the present invention provides a method of providing a service embodied in a component using a processing system, the method including causing the processing system to:

a) Receive a request for the provision of the service;
b) Generate a respective component instance in response to the received request;
c) Receive data to be manipulated;
d) Manipulate the data with the respective component instance; and,
e) Supply the manipulated data to an output.

The method can include causing the processing system to manipulate the data in accordance with at least one of:

a) A predetermined process; and,
b) Input commands received from an operator;

The method can include causing the processing system to:

a) Generate an agent associated with each input and output of the component;
b) Activate the agents, thereby causing the agents to perform at least one of:
    i) Cooperate with other agents to establish a connection;
    ii) Receive data from an agent;
    iii) Transfer data to an agent;
    iv) Cooperate with other agents to perform agent hand-off.

Each agent is typically formed from executable code executed by the processing system.

The method may be used the method of the first broad form of the invention.

In an eleventh broad form the present invention provides apparatus for providing a service embodied in a component, the apparatus including a processing system adapted to:

a) Receive a request for the provision of the service;
b) Generate a respective component instance in response to the received request;

c) Receive data to be manipulated;
d) Manipulate the data with the respective component instance; and,
e) Supply the manipulated data to an output.

The apparatus can be adapted to perform the method of the tenth broad form of the invention.

In a twelfth broad form the present invention provides a computer program product for providing a service embodied in a component, each component corresponding to a respective service for manipulating data, the computer program product including computer executable code which when executed on a suitable processing system causes the processing system to perform the method of the tenth broad form of the invention.

In a thirteenth broad form the present invention provides a method of allowing users to manipulate data, the method including:

a) Providing access to one or more components, each component representing a respective service for manipulating data;
b) Allowing user to define a combination of the components; and,
c) Causing the components to manipulate data in accordance with the defined combination.

One or more of the components can be implemented by respective entities.

The method can include charging a fee to users for the implementation of each component.

The method typically includes providing at least a portion of the fee to the respective entity.

The method may be implemented using a processing system, the method including causing the processing system to:

a) For each component, receive a component specification from a respective entity; and,
b) Provide details of one or more components to the user in response to a request, thereby allowing the user to request implementation of the one or more components, the details being determined from the specification.

The details can include a graphical representation.

The method typically includes providing a performance information for each of the components, the performance information representing the success of the component at performing the respective service thereby allowing users to select the components in accordance with the performance information.

The performance information typically includes at least one of:

a) An indication of the entity implementing the component;
b) An indication of the geographical location of the entity;
c) An indication of the duration for implementing the component;
d) An indication of a cost associated with implementing the respective component; and,
e) A rating, the rating being indicative of the success of the component.

The method being used the method of the first broad form of the invention.

In a fourteenth broad form the present invention provides apparatus for allowing users to manipulate data, the apparatus including a processing system adapted to:

a) Provide access to one or more components, each component representing a respective service for manipulating data;
b) Allow user to define a combination of the components; and,
c) Cause the components to manipulate data in accordance with the defined combination.

The apparatus being adapted to perform the method of the thirteenth broad form of the invention.

In a fifteenth broad form the present invention provides a computer program product for providing a service embodied in a component, each component corresponding to a respective service for manipulating data, the computer program product including computer executable code which when executed on a suitable processing system causes the processing system to perform the method of the thirteenth broad form of the invention.

**Brief Description of the Drawings**

An example of the present invention will now be described with reference to the accompanying drawings, in which: -

Figure 1 is a flow diagram outlining an example of the production of software in accordance with the present invention;

Figure 2 is a schematic diagram of an example of a processing system for generating computer executable code;

Figures 3A and 3B are a flow diagram of an example of the method of creating computer executable code using the processing system of Figure 2;

Figure 4 is a schematic diagram of an example of a web based system for generating computer executable code;

Figure 5 is a schematic diagram of an example of an end station of Figure 4;

Figure 6 is a schematic diagram of an example of an entity processing system of Figure 4;

Figure 7 is a flow diagram of an example of the method of having an entity provide a component to the base station of Figure 4;

Figure 8 is a schematic diagram of an example of a component properties dialog box;

Figure 9 is a schematic diagram of an example of an output properties dialog box;

Figures 10A to 10E are a flow diagram of an example of the method of creating computer executable code using the system of Figure 4;

Figure 11 is an example of a schematic representation that is presented to the user;

Figure 12 is an example of a component representation that is presented to the user;

Figure 13 is an example of the schematic representation of Figure 11 modified to include interconnections;

Figure 14 is an example of a schematic representation of the internal structure of the component of Figure 12;

Figure 15 is a schematic diagram of an example of a schematic representation for two interconnected components;

Figure 16 is a schematic diagram demonstrating the operation of the agents of Figure 15;

Figure 17A to 17E are schematic diagrams of a first example demonstrating the operation of hand off of agents;

Figure 18 is a schematic diagram of a second example demonstrating the operation of hand off of agents;

Figures 19A and 19B are schematic diagrams demonstrating the operation of agent bundles;

Figure 20 is a schematic diagram demonstrating the operation of a sequence of agent bundles;

Figure 21 is a schematic diagram demonstrating the operation of a debundle component;

Figures 22A to 22C are schematic diagrams of a first example of bundle and debundle use;

Figure 22D is a schematic diagram of a second example of bundle and debundle use;

Figure 23 is a schematic diagram demonstrating the operation of bundle reordering;

Figure 24 is a schematic diagram demonstrating the operation of bundle rake-out;

Figure 25 is a schematic diagram demonstrating the operation of the constructor;

Figure 26 is a schematic diagram demonstrating the operation of the constructor of Figure 25 to present components to the base station of Figure 5;

Figure 27 is a schematic diagram demonstrating the operation of the constructor of Figure 25 to present agents to other components;

Figure 28 is an example of a component representation of an "Add1" component;

Figure 29 is an example of an internal schematic of the "Add1" component of Figure 28;

Figure 30 is an example of a test schematic using the "Add1" component of Figure 28;

Figure 31 is an example of a component representation of an "Add2" component;

Figure 32 is an example of a component representation of an "Add3" component;

Figure 33 is an example of an internal schematic of the "Add3" component of Figure 32;

Figure 34 is an example of a component representation of an "Add5" component;

Figure 35 is an example of an internal schematic of the "Add5" component of Figure 34;

Figure 36 is an example of a component representation of a "Put Pixel" component;

Figure 37 is an example of an internal schematic of the "Put Pixel" component of Figure 36; and,

Figure 38 is an example of a test schematic using the "put Pixel" component of Figure 36.

Figure 39 is a schematic example of a hierarchical bundle;

Figure 40 is the component schematic of an example of a component adapted to modify the payload of the agent $A_1$ shown in Figure 39;

Figure 41 is the component schematic of an example of a component adapted to modify the payload of the agents $A_1$, $A_2$ shown in Figure 39;

Figure 42 is a schematic of an example of a component adapted to provide one-to-many interconnections to other components;

Figures 43A to 43E are a schematic example of the process of handing-off an agent

bundle;

Figures 44A to 44C are a schematic example of the process of staged construction;

Figures 45A to 45K are examples of component schematics for agent components;

Figures 46A to 46E are examples of component schematics for conditional components;

Figures 47A to 47M are examples of component schematics for construction management components;

Figures 48A to 48G are examples of the use of construction management components;

Figures 49A to 49B are examples of component schematics for conversion components;

Figures 50A and 50B are examples of component schematics for file components;

Figures 51A to 51L are examples of component schematics for logic components;

Figures 52A and 52B are examples of component schematics for protocol components; and,

Figures 53A to 53 D are examples of component schematics for string components.

**Detailed Description of the Preferred Embodiments**

An example of the process for producing computer executable code will now be described in outline with reference to Figure 1.

As shown, the first step is to determine the requirements for the computer executable code to be created at step 100. This is achieved by considering the functionality that needs to be implemented by the resulting computer executable code, as will be explained in more detail below.

Once the desired functionality has been determined, a number of components are selected that when combined in an appropriate manner will allow executable code having this functionality to be created. In this regard, each component corresponds to a respective service, which is used to manipulate data to thereby produce computer executable code. Thus, for example, the services can include processes such as the modification, removal,

movement or creation of data. This allows each component to contribute in some way to the formation of the computer executable code. The services may be performed automatically through the use of components formed from computer executable code, or the like. Alternatively the services may be performed manually, or through combination of manual and automatic implementation.

The level of complexity of the component services will vary as will be explained in more detail below. Thus, for example, simple components may operate to erect one or more bytes in a binary file, which are then, used in forming CPU instructions, whereas more complex components may operate to erect several CPU instructions simultaneously.

In order to achieve this, each component is adapted to receive data via one or more respective inputs, and then perform manipulations of the data as required. Similarly, the majority of components will also include one or more outputs for allowing manipulated data, or other information to be output.

In use, the components interact with each other by transferring data therebetween. Thus, for example, the output of one component may be connected to the input of another component, to allow two services to be performed in sequence. Combining appropriate ones of the more basic level components in a hierarchical structure can also be used to allow more complicated services to be implemented as a collection of more basic services.

Examples of components are set out below.

Accordingly, at step 120, a combination of the selected components is defined which will allow the computer executable code to be created. In particular, this specifies how the components should be interconnected via the inputs and outputs, such that when the

services provided by the components are implemented at step 130, the interaction results in the generation of the desired computer executable code.

It will be appreciated that the implementation of this technique can be achieved in a number of ways. However, in its broadest form, this process can be performed using a single processing system an example of which is shown in Figure 2.

In particular, the processing system 10 generally includes at least a processor 20, a memory 21, and an input device 22, such as a keyboard, an output device 23, such as a display, coupled together via a bus 24 as shown. An external interface is also provided as shown at 25, for coupling the processing system to a store 11, such as a database.

In use, the processing system is adapted to allow details of available components to be stored in the database 11. A user can then define a combination of selected components, allowing the processing system 10 to generate the computer executable code. From this, it will be appreciated that the processing system 10 may be any form of processing system such as a computer, a laptop, server, specialised hardware, or the like.

The manner in which the processing system 10 may be used to generate computer executable code will now be described with reference to Figures 3A and 3B.

In particular, this example describes a situation in which a number of components are provided in the database 11, which may be implemented automatically in order to perform the required services. Accordingly, this allows a user to generate computer executable code using the processing system 10 alone.

In order to achieve this, the user determines requirements for the computer executable code to be created at step 200. At step 210 the user provides a component request to the

processing system 10. The request may be any form of request, but will typically be in the form of a request for details of the components stored in the database 11.

In this case, the details of the components are stored in the form of component specifications, which indicate at least the service performed by the respective component. The component specifications may also include input and output specifications providing details of the type and/or form of data that each input/output is adapted to receive/provide. The component specifications may be any one of a number of forms depending on the implementation of the system, and therefore may be provided as, or at least include a graphical representation, text data, operational parameters or the like.

Accordingly, at step 220, the processing system 10 accesses the component specifications stored in the database 11, and uses this to provide an indication of one or more of the components to the user at step 230. The indication may be in any one of a number of forms depending on the implementation, and may therefore include graphical or textual representations, or the like. It will therefore be appreciated that the indication may be all or part of the specification itself.

The indication of the one or more components may be performed in such a manner that the user can specify one or more services to be performed in the request, with the processing system 10 responding to only provide details of those components able to complete all or part of the specified services.

This allows the user to select appropriate ones of the components and provide a component selection to the processing system 10 at step 240, thereby indicating the one or more selected components.

The processing system 10 uses this information to generate a component indication

including an indication of the selected components at step 250. The component indication may be in a graphical form or may be in the form of a list specifying the components selected. This can therefore optionally be presented to the user on the display 23, or the like.

At step 260 the user determines at least two of the selected components to be connected. It will be appreciated that in order to perform the connection, it is important that the format and/or type of data handled by the respective components to be connected is compatible.

Thus, for example, if the output of a first component is coupled to the input of a second component, then it is important that the service of the second component is able to operate on the data output by the first component. This process may be performed manually by observation of the input and output specifications, or alternatively may be performed with the assistance of agents, as will be explained in more detail below.

The user then provides a connection indication representing the desired connections to the processing system 10 at step 270. This may be achieved by providing details of each input and output of the two different components to be connected, however alternatively the mechanisms may also be used such as manipulation of graphical representations or the like.

In any event, at step 280 the processing system 10 operates to interconnect the components in accordance with the connection indication.

At step 290 the user determines if more connections are required and if so returns to step 270 to define further connections. If not, the process moves on to step 300 at which point the user determines if additional components are to be selected. If so, the process returns

to step 210, allowing steps 210 to 290 to be repeated.

Otherwise, the user optionally reviews the defined component interactions to determine if the executable code is to be constructed at step 310. In particular, this is generally performed to assess the expected performance of the code, the construction time, or the like, to determine if construction of the code through the specified code is feasible. Other factors that may be assessed include the expected cost, which may be relevant if the user has to pay a fee for the implementation of each component.

If it is determined that the specified component interactions are not acceptable for any reason and that the code is not to be built at step 320, then the process ends at step 330. It will be appreciated that as an alternative option, the user may return to any previous step in the process and revise the specified component interactions, for example through the removal, modification or addition of the components, or the component interactions.

Otherwise the user causes the services defined by the interconnected components to be performed at step 340.

The manner in which the components are implemented will vary depending on the respective component form. As described above, in this example, each of the components is formed from computer executable code stored in the store. Accordingly, when a component is to be executed the code can be downloaded on to the processing system 10 and executed in the normal manner.

Accordingly, each component will operate to manipulate data stored either in the memory 21, or the database 11, in turn, in accordance with the specified interconnections.

Thus, for example, initial data may be supplied to the input of a first component, which

then operates to manipulate the data in accordance with a service defined therein. When this has been completed, the manipulated data is provided at the first component output. The data will then be transferred to the input of a second component, allowing the service defined by the second component to be performed.

A similar process will occur for components having multiple inputs and/or outputs.

It will be appreciated that variations may arise for different implementations. Thus, for example, the components may not all be implemented by the processing system 10 itself, and instead may be implemented remotely on other processing systems, as will be explained in a further example below. Similarly the components may not all be performed automatically, and may require the user to provide inputs, transfer data, and perform some data manipulation.

This can either be intentional arising as a result of the manner in which the service associated with the component is implemented. Alternatively, manual intervention can be unintentional, if for example a fault occurs in the implementation that requires user input to resolve an issue, such as the requirement to transfer incompatible data formats between components.

This is repeated for all components, until the computer executable code is generated in the memory 21. Once generated, the code can be output to the user, allowing the code to be implemented on other processing systems, in the normal way.

It will be appreciated that this is feasible because the components interact both horizontally, and vertically in a hierarchical fashion. Accordingly, complicated services can be performed easily by combining simple components in an appropriate manner.

Accordingly, it will be appreciated that through the creation of basic components, which are then combined in appropriate manners, complicated data manipulations can be performed, in turn allowing computer executable code having a complex functionality to be developed.

It will be appreciated that the process described above with respect to the processing system 10 may be implemented using a number of different architectures. Thus, for example, the system can be implemented using a distributed web based system, or the like, with user accessing facilities provided by a central processing system 10 via the Internet, or another communications network.

An example of this will now be described in more detail with respect to Figure 4.

In particular, in this example, one or more central processing systems 10 (two shown in this example for clarity purposes only) are provided at a base station 1, which is coupled via a communications network, such as the Internet 2, and/or a number of local area networks (LANs) 4, to a number of end stations 3.

In use, the components may be provided at, and implemented by, the processing system 10, as described above. Alternatively, the components may be provided by one or more respective entities, each of which operates one or more respective entity stations 5, which are also coupled to the Internet 2, and/or the LANs 4, as shown. In this example, each entity station 5 is formed from an entity processing system 15, coupled to a store, such as a database 16, as shown.

In use, users of the system can use the end stations 3 to communicate with the base station 1 to thereby obtain the provision of services embodied in suitable components.

This may be achieved in a number of manners however in this example, access of the services is provided through the use of web pages, although this is for illustrative purposes only. In order to achieve this, each end station 3 is therefore formed from a processing system that is adapted to access web pages and transfer data to the end station 1, as required.

An example of a suitable end station 3 is shown in Figure 5. As shown the end station 3 includes a processor 30, a memory 31, an input device 32, such as a keyboard, or the like, an output device 33, such as a display, which are coupled together via a bus 34. The processing system is also provided with an external interface 35 for coupling the end station 3 to the Internet 2, or the LAN 4, as required.

In use, the processor 30 is adapted to communicate with the processing system 10 provided in the base station 1 via the communications networks 2, 4 to allow the processing system services to be accessed. Accordingly, it will be appreciated that the end stations 3 may be formed from any suitable processing system, such as a suitably programmed PC, Internet terminal, lap-top, hand-held PC, or the like, which is typically operating applications software to enable data transfer and in some cases web-browsing.

The components can be implemented either at the processing system 10 itself, or at one of the entity processing systems 15, depending on the nature of the component and the service provided therein.

The entity processing system 15 must therefore be able to communicate with the processing system 10 via the communications networks 2, 4. In order to achieve this, the entity processing system 15 would generally be similar to the processing system shown in Figure 6.

As shown the entity processing system 15 includes a processor 40, a memory 41, an input device 42, such as a keyboard, or the like, an output device 43, such as a monitor, which are coupled together via a bus 44. The processing system is also provided with an external interface 45 for coupling the entity station 5 to the Internet 2, or the LAN 4, as well as the database 16, as required.

In use, the processor 40 is adapted to allow the entity to perform the services encapsulated in respective components. Accordingly, it will be appreciated that the entity stations 5 may be formed from any suitable processing system, such as a suitably programmed PC, Internet terminal, lap-top, hand-held PC, or the like. Typically however, as the services are data intensive, the entity processing systems 15 will be formed from servers, or the like.

To allow components to be implemented by the entity stations 5, whilst still allowing users of the end stations 3 to access the services provided therein via the base station 1, it is typical for details of the components to be stored in the database 11, in the form of component specifications. The component specifications may be in any one of a number of forms, and may include graphical representations, or the like. However, in general the component specifications include sufficient information for a user to determine the service embodied by the respective component.

The manner in which computer executable code may be created will now be described with reference to Figure 7.

In particular, at step 400 the entity determines a manner of providing a respective service. This may be achieved in a number of ways and will depend on the respective service and the manner in which the entity wishes to provide the service.

Thus, for example, the entity may provide the service manually such that the entity receives data at the entity station 5, modifies the data using the entity processing system 15, and then returns the modified data to the processing system 10 or the end station 3, all under control of the user.

Alternatively, the process may be performed by computer executable code, executed by the entity processing system 15, in which case, the entity must first determine the necessary executable code.

A combination of manual and automatic processes may also be used. Furthermore, data may not be returned to the processing system 10 or the end station 3, but instead may be transferred to another one of the entity stations 5 for manipulation by another service embodied by a different component.

As a further option, the entity may provide a service in the form of a compound component. In this case, the entity effectively defines a combination of previously existing components, which when combined define a component allowing the required service to be performed. In this case, the entity station 5 will be adapted to hand-off implementation of the components contained within the compound component to other ones of the entities, such as through other entity stations 5, and/or the base station 1, as required.

In any event at step 410 the entity defines a component encapsulating the provision of the service using the entity station 5. In order to achieve this, the entity processing system 15 will generally be provided with applications software that aids the entity in this process. In particular, the software will prompt the entity to provide information that will be required by the processing system 10 to allow the functionality provided by the respective component service to be determined by a user. Thus, for example the entity

may be presented with a dialog box including fields defining the types of information that are required in order for users to determine the operation of the component.

In general, the required information includes at least component, input and output specifications. In particular, the component specifications are used to provide information regarding the service provided by the component, together with information regarding the component author, implementing entity, or the like. The component specification also includes sufficient information to allow the processing system 10 or the end station 3 to access the services provided by the component.

Accordingly, the component specifications typically include at least:

- Manufacturer ID – used to identify the entity providing the service
- Component ID – used to identify the respective component
- Location information – used to identify where the component is implemented
- Description – an indication of the service provided by the component

This information may be provided for example through the use of a properties dialogue box shown for example in Figure 8. The properties dialogue box will prompt the entity to provide information such as the component name, the component description, the author, the address, report number or the like.

The applications software installed on the entity processing system 15 can also be used to generate any identifiers that may be required. In particular, it is generally necessary to generate identifiers to allow both the entity, and the component to be uniquely identified.

Furthermore, an entity station 5 may be implementing the same component simultaneously for a number of different code generation projects. In this case, several different component instances will exist, with each component instance being applied to

each respective code generation project. Accordingly, in this case, it is also necessary to generate respective identifiers allowing each component instance to be uniquely identified.

In addition to this, the entity also provides input and output specifications, which are used to indicate the types and/or formats of data that can be received by the component inputs, or output from the component outputs. This is important for ensuring that components are able to communicate with each other, by transferring data from the output of one component to the input of a subsequent component.

In this example, control of this communication is achieved using agents, which are software applications executed at the location at which the respective component is implemented. The agents operate to negotiate between available data types and formats specified in the input and output specifications, to allow respective components to communicate directly.

In general, the input and output specification may also include details of the manner of operation of the respective agent. Accordingly, the details may be provided through the use of a dialog box that prompts the entity for details regarding the respective input and/or output and associated agent. An example of a dialog box for an output is shown in Figure 9.

The operation of the agents will be described in more detail below. However, it will be appreciated that the entity also operates to construct agents when encapsulating the service as a component.

Accordingly, at step 420 the entity processing system 15 operates to store the generated component, input and output specifications, and agents, typically in the database 16. The

entity station 5 is then used to access the base station 1 at step 430, allowing details of the component, input and output specifications, to be transferred to the base station 1 at step 440, for storage in the database 11 at step 450.

It will be appreciated that if the component is self contained, the entire component may be downloaded to the database 11, for storage thereon, in which case there is no requirement to store any information at the entity station 5. This allows the component service to be implemented by the processing system 10 automatically, as described above for example with respect to Figures 3A and 3B. Alternatively, the component may be transferred to the end station 3 for implementation thereon. These techniques will generally result in the manner of implementation of the services to be made publicly available.

Typically however, the entity will wish to retain at least some form of control over the operation of the component for a number of reasons, in which case the component service may be implemented at the entity station 5.

This is also generally required if the service implementation requires manual input from the entity, but may also be desirable for other reasons.

Thus, for example, this allows the entity to monitor use and operation of the component, as well as making it easier for the entity to adjust and/or modify the operation of the component to improve its efficiency. Furthermore, this allows the entity supplying the service to provide only the manipulated data, or another output, and not divulge method used to implement the service. This allows the implementation of the service to be retained as a trade secret, specialised knowledge or the like.

In any event, as the system is adapted to handle a large number of components, it is

generally undesirable to have all these located at the base station 1, as the database 11, and processing systems 10 would rapidly become over used.

Accordingly, the components are usually implemented at the entity stations 5, with details of the specifications and the agents being transferred to the base station 1, to allow users of the end stations 3 to select the components for use. In particular, when the users of the system select components in this fashion, it is transparent to the user whether the component itself is actually provided at the base station 1 or whether the component is provided at an entity station 1. This is because all the specifications and agent details needed to access the entity station 5 providing the respective service are stored in the base station 1.

A detailed example of the manner in which a user uses the base station 1 to produce applications software will now be described in more detail, with respect to the flow chart set out in Figures 10A to 10E.

Accordingly, as shown at step 500 in Figure 10A the first stage is for a user to determine the requirements of the computer executable code to be created. At step 510 the user then accesses the base station 1 using the end station 3.

At step 520 the user selects a component search using the end station 3 and this causes the processing system 10 to provide details of available components based on component specifications stored in the database 11, at step 530. In particular, the processing system will typically allow users to search through categories of components, with the categories defining different forms of functionality. This allows users to rapidly locate components that are suitable for performing required services.

At step 540 the user reviews the component properties and selects one or more

components. This may be achieved in a number of ways, although typically the user will be presented with navigable lists that provide at least a component title and brief additional description of suitable components. The user can then select a respective one of the components allowing further details to be provided, and ultimately, the selection to be made.

The details may be provided for example through the use of the properties dialogue box similar to that shown for example in Figure 8. In this case, the details include information such as the component name, the component description, the author, the address, report number, or the like, and will be determined directly from the component specifications stored in the database 11.

At step 550 the end station 3 (or alternatively the processing system 10) stores an indication of the selected components. This may be achieved in a number of manners depending on the implementation. Thus, for example, the end station 3 typically generates component data, which is stored in the memory 31, the component data including an indication of each component selected by the user. Alternatively, however, the processing system 10 may generate the component data and store it in the database 11.

At step 560 the end station 3 (or alternatively the processing system 10) generates a schematic representation, including representations of the components so far selected. The schematic representation is used to allow the user to define the component interconnections, as will be described in more detail below. In particular, this allows the user of the end station 3 to visualise the components and how these will need to interact with each other to produce the computer executable code.

The schematic representation includes a representation of each of the components

selected. The component representation is generally generated by the entity and transferred to the base station 1 as part of the component specifications. When the user selects a respective component, the corresponding component representation is transferred from the base station 1 to the end station 3, and added to the schematic representation, as required.

It will therefore be appreciated that the indication of the component stored by the end station 3 may be in the form of the component representations.

An example of a schematic representation is shown in Figure 11. As shown, the schematic representation is displayed in a schematic window 50, and in this example, includes four component representations 51, 52, 53, 54. Each of the components has a number of inputs and outputs, as shown generally in the component representation at 51A, 51B 51C, .....

If the user selects a respective one of the components shown in the schematic representation, the user is presented with a single component representation, an example of which is shown in Figure 12. In particular, Figure 12 shows a component display screen 60 including a component representation 52 having a number of input and output representations 52A, 52B ...., 52G presented thereon.

The component display screen also includes a number of window selection tabs 61, which allow the user to navigate between the component window 60 shown, the schematic window 50 mentioned above, and an auto select window.

An agent window 62 is also provided, which displays details of a selected input or output agent (in this example agent 52G), obtained from the respective input and/or output specification.

In use, the user can navigate around the component and schematic representations to allow various information regarding the components to be provided. Thus, for example, by selecting the component representation 52, this can allow the properties of the corresponding component to be displayed, as shown for example in Figure 8. Similarly, by selecting a respective one of the input and/or output representations, details of the respective input or output will be displayed in the agent window 62. These details will typically be provided by displaying an input or output dialog box, similar to the one shown in Figure 9, as appropriate.

In any event, the user reviews the presented schematic representation and determines if further components are required at step 570. If it is determined that more components are required at step 580, the process returns to step 520 to allow the user to return to the component search tool and select more components using the end station 3. Representations of these components can then be added to the schematic representation as required, for example using drag and drop techniques.

Once the required components (or at least some of the required components) are selected, through the placement of corresponding component representations on the schematic representation, the user determines component inputs and outputs that are to be connected at step 590.

In order to ensure that the components may interact successfully, the user will typically check at this point whether the input and output that are to be connected are compatible at step 600. In particular, the user checks whether the input and output can handle any common data types and/or formats. This information can be determined by examination of the input and output details determined from the input and output specifications.

If the user determines that the input and output cannot be connected at step 610, the process returns to step 520 to allow one or more alternative components to be selected.

Otherwise, the user selects a connection tool and operates to generate a connection representation between the input and output of the respective component representations on the schematic representation, at step 620. An example of this is shown in Figures 13 and 14. The end station 3 interprets the connection representation as a connection between the respective input and output, and generates connection data representing the connection.

In particular, Figure 13 shows the schematic representation of the component representations 51, 52, 53, 54 of the component representations shown in Figure 11, with the components being interconnected using the connection representations shown generally at 65. In particular, in this example, the component representation 52 is coupled to a duplicate component representation 51, a BNE addressed component representation 53, and a build component representation 54, as shown.

Figure 14 shows that the component representation 52 corresponds to a compound component formed from a number of sub-components. These sub-components are in turn represented as a LDAA component representation 70, a DECA component representation 71, an STAA component representation 72, and two build component representations 73, 74, interconnected by the connection representations 65, as shown.

The combination of components represented by the schematic shown in Figure 13 allow computer executable code forming a decrement counter to be produced. However, it will be appreciated that this example is provided to demonstrate the operation of the schematic representation and the actual functionality implemented is not important. Additional examples are described in more detail below.

In any event, the user can select a respective input and output on the schematic representation, and then draw on a connection representation between the inputs and outputs at step 620 thereby defining a connection between the respective input and output.

In this example, neither the end station 3 or the processing system 10 operate to examine the validity of the connections, and in particular does not determine whether data can successfully be transferred from the output of the first component, to the input of the second component. However, it will be appreciated that checking by the processing system may be performed in some implementations.

In any event, in this example, the end station stores an indication of the created connection in the form of connection data at step 630.

The user then reviews the schematic representation and determines if further connections are required at step 640. If it is determined that further connections are required at step 650, the process returns to step 590 to allow further connections to be defined in the manner described above.

Thus effectively, the user will use the drawing tool to define all the connections required in the schematic representation. This will typically require that each input and output of each component is coupled either to another output or input.

If it is determined that no further connections are required for the components in the schematic representation at step 650, the user reviews the schematic representation and determines if more components are required at step 660. This allows the method to return to step 520 so that more components may be included, if it is determined that more components are required at step 670.

Thus, the user can effectively select two or more components and operate to interconnect these, before going back to select further components.

If it is determined that no further components are required at step 670, the user indicates that the computer executable code is to be constructed at step 680.

At this point, the user may optionally review the schematic representation and determine if the computer executable code is to be generated, as outlined above for example with respect to steps 310 to 340 in Figure 3B. In particular, the user will generally be presented with information regarding the overall code generation process, such as an indication of the overall cost, time to build, resource usage, resultant performance, or the like.

This is typically generated by having the end station 3 and the processing system cooperate to determine the relevant information. Thus for example, the end station may transfer an indication of the schematic to the processing system to allow the relevant values to be determined, or the like.

In any event, the allows the user to assess whether they are satisfied with the construction process defined by the respective schematic representation, and therefore whether they wish to proceed with construction of the computer executable code.

The construction of the computer executable code by implementation of the services defined in the schematic representation is known as a build process. This is implemented through the use of agents, which operate to allow the components to interact, as will now be explained in more detail.

In particular, upon receiving instructions to proceed with the build process at step 680, the end station 3 accesses the component and connection data at step 690. At step 700, the end station uses the component data to determine the components to be used in generating the computer executable code.

It will be appreciated that the build process may alternatively, or additionally be performed by the processing system 10, one or more of the entity stations, other processing systems, or a combination of the above depending on the respective implementation. However, the remainder of this example will be described with reference to the build process being performed by the end station 3.

At step 710 the end station 3 generates a purchase order corresponding to each component to be used. In particular, the purchase order is adapted to be sent to the entity providing the respective service, via a respective entity station 5, to request the provision of the services associated with the respective component. In general each purchase order will include at least the following information:

- Manufacturer ID
- Component ID
- Build ID – used to identify the respective build instance
- Restrictions – an indication of any restrictions placed on the implementation by the user

It will be appreciated that whilst the above describe the use of purchase orders, these are not essential, and alternative techniques for ordering the implementation of services associated with respective components may alternatively be used.

At step 720 each purchase order is sent to the respective entity, allowing each entity to determine if it is capable of performing the respective service at 730. Thus for example,

an entity may become unavailable due to implementation problems such as faults with the computer executable code or unavailability of an individual performing the service manually, or the like.

It will be appreciated that in the event that a component is formed from a number of sub-components, the inability of an entity to implement the component may arise from the failure of one or more of the sub-components, which in turn may be the responsibility of other entities.

If an entity cannot perform a service, whether this is due to a problem with the respective component itself, or any associated sub-components, an indication of this is transferred to the end station 3. Accordingly, at 740 the end station 3determines if all components can be performed. If not, the user updates the schematic representation by selecting one or more alternative components at step 750. Thus, for example, the process can return to step 520, to allow different components to be selected.

If an entity can perform a requested service, an indication of this will also be transferred to the end station 3, indicating a respective component instance ID, which uniquely identifies the component instance that has been assigned to perform the task. This is important as it will be appreciated that in general, each processing system 15 might be performing the same component simultaneously for a number of different build processes. Accordingly, the use of a component instance ID ensures that data is transferred to the correct component instance for processing.

If it is determined that all the services encapsulated by all of the components can be performed, end station 3 (or the processing system performing the build) determines respective inputs and outputs that are to be connected during the process at step 760. This is determined in accordance with the connection data.

At step 770 the end station 3then activates each agent associated with each input and each output to be connected.

In particular, the agents are activated in a predetermined sequence as will be described in more detail below. As an agent is activated, the agent determines details of the respective data formats and/or types from the respective input/output specification at step 780. At step 790 the agents then compare the determined data formats/types.

In particular, the agents of the respective input and output cooperate to compare the respective data formats/types to determine if there are any data formats/types in common. In this regard, it will be appreciated an input and output can only successfully communicate if both the input and output are able to handle a common data format and/or type.

If it is determined that there are no data formats/types in common at step 800 the process proceeds to step 810 at which points the agents determine that the components can not be interconnected. This will occur for example if one of the components is outputting data in a first format whilst the other component needs to accept the data a second format. If this occurs, the build process is halted and the user informed. This allows the user to take corrective measures to allow the build process to continue. This may be achieved, for example by adding in additional components or agents, or my manual manipulation of the data, to allow the error to be corrected. Alternatively, the build process can be terminated such that the software cannot be constructed.

In any event, if the respective input and output have data formats/types in common, then at step 830 the agents indicate that the respective inputs and outputs are ready to communicate. Once all the connections are ready at step 840 then the processing system

10 activates the components at step 850. In particular, the components will provide the respective services defined therein by having the entity stations 5 interact with data. The data may be manipulated before being transferred to the end station 3, or another one of the entity stations 5, as required by the defined schematic representation.

In general, each component will be implemented at the respective entity station 5. In order to achieve this the data to be manipulated will be downloaded from the end station 3, the base station 1 or another one of the entity stations 5, to the respective entity station 5. This will be achieved by transferring the data to a specific port or the like on the processing system 15, as indicated in the component specification. The data will be transferred with the component instance ID to ensure that the correct component instance is used to manipulate the data.

It will be appreciated however that this may be achieved using other techniques, such as providing each content instance at a respective port, and transferring the data to the respective port.

In any event, when a respective component instance receives the data to be manipulated, the respective component will interact with the data modifying the data as required before providing the modified data at one or more of the output ports.

The data will then typically be transferred to the base station 1 or the end station 3 for temporary storage in the memory 21 or the database 11, before being transferred to the input of the next component. Alternatively however the data provided at the output port of a component at one of the entity stations 5 could be transferred directly to another entity station 5, for subsequent manipulation by another component.

It will be appreciated that during this process, data may also be manipulated

simultaneously by several different components depending on the format of the component specification.

Furthermore, it is usual for the base station 1, the end stations 3, and the entity stations 5 to be effectively interchangeable or implementable on a common processing system in the examples outlined above. Accordingly, the processing system 10, 15 and the end station 3 will generally execute applications allowing the functionality of each of the base station 1, the end station 3, and the entity station 5 to be implemented. For example, this allows an entity to use the entity station 5 to create executable code in the manner outlined above for the end station 3, and vice versa.

Thus, for example, an entity may have a number of processing system, some of which operate as entity stations 5, and some of which operate as end stations 3, depending on the functionality required at the time. Thus, for example, the entity may be providing a number of component instances, the implementation of which is distributed across the processing systems. In this instance the functionality provided by the processing systems will be equivalent to either or both of the end stations 3 and the entity stations 5, as required.

It will be appreciated that performing the implementation of components will typically require a support structure, and it is therefore common for the entity to have an infrastructure in place including a number of end stations 3 that will be used in supporting the implementation of the service, as well as to allow software to be generated.

In the case of an entity providing a service, the entity would typically have a number of entity stations 5 that will be automated. However, if an exception, or other error occurs, such that the component cannot complete the service, then the entity station 5 will hand-

off or transfer the component to another entity station 5 that is operated by an individual. This allows the individual to provide manual feedback to allow the exception or error to be resolved, if possible. Otherwise, an indication that the problem cannot be resolved will be returned to another component or entity within the system. Thus, if the problem occurs with a sub-component an indication of the problem will initial be returned to the parent component. This will continue with the exception being passed up the chain until it can be resolved.

Some of the features of the implementation described above, such as the nature and operation of the agents is described in more detail below.

Practical Implementation

It will be appreciated from the above that the base station 1 allows services provided by a number of different entities, typically at respective entity stations 5, to be accessed centrally by a number of different users. This allows components provided by entities to be reused a large number of times in the creation of numerous different software applications.

In general, the system will be implemented as a forum that provides users with access to the different services. The forum may be implemented using a single base station, as in the example described above. However, persons skilled in the art will appreciate the forum may be implemented using a number of base stations, and a number of associated processing systems, with the forum being distributed between the base stations and the user end stations 3.

The following description therefore focuses on the implementation of the system using a forum, although the techniques are equally applicable to any implementation, such as the use of a single base station.

In use, it is typical for each entity to define a fee associated with each component. This fee corresponds to a fee payable by users of the forum, for the use of a respective component instance service. Thus, the users pay one or more fees to each entity in return for the provision of one or more services provided by the entity.

This allows the entities to charge a fee for the provision of the respective services, thereby allowing the entities to obtain income to recoup the investment made in the development of the respective components. This in turn allows entities to specialise by providing, and obtaining financial return for, specific well-defined services.

This is in contrast to current software construction techniques in which entities typically only obtain financial benefit by constructing all the required executable code for entire software applications.

This in turn allows entities to focus on optimisation of specific service provision, rather than trying to create an entire software application that can only function adequately.

By having the forum provide users with access to a number of components, provided by different entities, and which provide similar services, this will force entities to compete against each other to provide similar services to the user. The resulting market forces will therefore drive competition between the entities, thereby forcing each entity to improve the provision of its respective service.

In particular, users will tend to select components that are deemed to be more successful. As a result, entities compete with each other at the component level to provide more and more successful components. This allows the entities to invest more time and money in improving the implementation of the specific components, whilst recouping the

investment as more successful components will be implemented a larger number of times.

In this regard, components may be deemed to be more successful if they are cheaper, faster, result in more optimal code, or the like, when compared to other components offering the same service.

From this, it can be seen that market forces and direct competition at a specialisation level will lead to improvement in each service provided through the forum. Thus, each component at every level within the hierarchical structure will be optimised resulting in the generation of optimal code with no bugs or other errors. This reflects an industrialised approach to software creation in which competition occurs directly at the specialisation level.

In order to help competition within the forum, the user will be provided with information to allow an assessment of which are the best components for use in constructing the respective application. The user can then select components in accordance with a wide variety of factors including, for example:

- The entity performing the respective service;
- The cost;
- The location of the entity performing the respective service;
- The popularity of the component;
- The data format/types that can be received by or output from the component; and,
- Ratings given to the component by previous users or the forum.

In this regard, the forum will generally provide a rating system allowing users to rate the effectiveness of components. Ratings can be determined statistically, for example by

determining the number of build faults that occurred for each respective component, by user feedback, or by testing of the components by the forum itself.

It will be appreciated that whilst market competition through the use of reviews or the like exist, this is normally provided with respect to entire software code. In contrast, the review and rating in this instance is performed at the component level thereby forcing the improvement of individual components, as opposed to entire software applications.

It will be appreciated that other factors may also be used in judging the success of components.

In any event, in order to remain competitive, each entity will focus on providing well-defined, efficient service implementations, allowing vastly improved software to be created. It will be appreciated that as components improve so will entire software applications thus the development of the new software generation technique will lead to rapid improvement in software applications.

In order to allow the operators of the forum to make a profit, it will also be typical for at least a portion of any fees charged by the entities, to be provided to the operator of the forum, allowing the operator to obtain profit based on usage levels of respective components. However, alternatively, subscription charges or the like could be applied to individuals wishing to use the system, or entities wishing to submit components to the forum.

Further details of example of the implementation of the processes outlined above will now be described in more detail below.

Agents

Agents are the only form of inter-component communication. Agents are responsible for providing and gathering all the information a component needs to complete the service it embodies. An agent is generally formed from a simple piece of executable code with limited functionality.

In particular, when the forum, and in particular the end station 3 sends out purchase orders to the entity stations 5, the purchase orders contain agent connections specifying how the agents of the respective components should interconnect.

The reception of a purchase order causes the entity station 5 to implement a constructor, which is described in more detail below. The constructor generates a new component instance, and corresponding agents that are capable of finding and connecting to the agents of other components. The agents only ever connect to (communicate with) other agents.

Whilst the agents are themselves simple, processes called bundling and hand-off allow agents to exhibit complex behaviour and powerful information providing and gathering capabilities.

The hand-off mechanism terminates an agent to agent transaction and opens another. This is most useful when a component is using the agent of a sub-component as if it were an agent on the component itself. Worked examples help to clarify the hand-off procedure and an example of hand-off to a sub-component is presented.

Bundling is a recursive mechanism by which multiple agents related by a specific purpose can be treated as a single simple agent. Worked examples help to clarify the

bundling and debundling mechanism and an example of bundling and debundling components are presented.

In particular, an example will now be described with reference to Figure 15, which shows a schematic P having components X and Y connected by agents X1 and Y1.

In order to specify the address of a particular agent for a particular component instance, it is necessary to be able to identify the agent uniquely. Accordingly, for the purposes of this example, the component X is manufactured by an entity having a manufacturer ID IDx, and component Y is manufactured by a an entity having a manufacturer ID IDy.

When schematic P is laid out, the symbols for components X and Y are downloaded, arranged and connected in the schematic representation P using the method described above with respect to Figures 10A to 10E. As the schematic is constructed, each component is given a unique label, as shown at U1 and U2. These labels allow schematics with more than one component of the same type to reference the correct instance of that component.

Associated with the respective component representations are the component specifications, including the manufacturer ID, and part number. This allows the forum to generate and send out the purchase orders to the corresponding entities. The component symbols are also associated with input and output specifications detailing the agents, which for this example is exactly one for each component.

Before the schematic can be built, the user creating the schematic P must be specified so that the entities IDx, IDy can bill the user. In this example, the user is given an identifier IDp. Once this information is contained in the schematic it is ready to be built.

The process of building a schematic results in a number of entities being contracted with purchase orders. Thus the decision to build will incur costs and contractual responsibility. A mistake in the schematic may result in a bad build wasting time and money.

Assuming the build is to proceed, the next step is to submit schematic P to the builder program. The builder program interprets the schematic and compiles and issues purchase orders for each component in the schematic. The purchase orders for the component X would contain the following information:

- Base station identifier 1
- Schematic identifier P
- X component label U1
- Entity identifier IDx
- Entity part number X
- Component X agent connection details, including:
    - Entity identifier IDy
    - Entity part number Y
    - Component Y agent number
    - User's identifier IDp
    - Schematic identifier P
    - Y component label U2

The purchase order for the component Y would include similar information.

Should the component being purchased have more than one agent, then each agent must have separate connection details included with the purchase order specifying the agent's target.

When the entity IDx receives the purchase order from the user IDp, the entity IDx creates an instance of its component to satisfy the requirements of the purchase order. This is achieved using a constructor, the operation of which will be described in more detail below.

After the provision of the purchase orders, the respective instance of the component X now has the information it needs for the agent X1 to communicate with the agent Y1. Accordingly, the agent X1 connects to the entity station 5 of the entity IDy and requests the agent Y1, using the information received in the respective purchase order, as outlined above.

Agents X and Y connect and then proceed to authenticate using the information already known about each other.

After the connection and authentication is complete components X and Y can negotiate to determine the information needed for each to complete construction of their respective components.

Thus, for example, in the event that the agents are adapted to handle the data types shown in Figure 16, the agent X1 can handle integers INT, characters CHAR, and floating point numbers FLOAT, whereas the agent Y1 can handle double inputs DOUBLE, floating point numbers FLOAT, and dates DATE. Accordingly, the agents will determine the component X must provide the output in the form of floating point numbers FLOAT.

From the above it will be appreciated that the agents are the only form of inter-component communication. As a result the agents must be able to communicate via the communications networks 2, 4.

A number of features of the implementation of the agents will now be described.

Hand-Off

In addition to agents making a static connection there exists an agent hand-off mechanism. The agent hand-off is mechanism allows an agent to agent connection to terminate with one of the agents reconnecting to yet another agent.

This behaviour provides a means by which agents can exhibit seemingly complex behaviour with simple agents. This is achieved by a component performing some simple information transaction then handing the agent-off to another component to negotiate further. With only these simple transactions a complex overall transaction can occur with the seemingly single agent.

In particular, handoff allows a component to present a service that internally is made up of carefully crafted arrangement of sub-components. Thus although a component presents a number of agents and appears to perform a given service, in actual fact the service is supplied by many sub-components which have at least some of their agents satisfied by a handoff from a parent agent. In order to achieve this an agent of the parent must first connect to some outside component, then order that agent at the other end to connect to a sub-component.

A walk through of the handoff mechanism demonstrates the steps involved in the handoff process.

Figure 17A represents a schematic involving a handoff. In Figure 17A agent A1 of components C1 and agent A2 of component C2 connect as normal, however C2 has sub-component C3 as shown in Figure 17B.

The component C2 intends to handoff the agent A1 to the agent A3 of the sub-component C3 as shown in Figure 17C.

However to simplify the agent protocol, it is assumed that each agent only connects to one other agent at a time. Thus the agent A2 could not connect to the agents A1 and A3 at the same time. To allow the component C3 to be built and access to the agent A3 to be gained, a temporary agent A2b is created as shown in Figure 17D.

This has the benefit of allowing the component C3 to be built and the agent A3 to connect to an agent providing means for the component C2 to communicate to the component C3. When the agent A1 is connected to the agent A2 and the agent A2b is connected to the agent A3, the component C2 can direct the agents A2 and A2b to terminate and cause the agent A1 to reconnect to the agent A3

Thus, the component C2 uses the local agents A2 and A2b to communicate the hand-off order to the agents A1 and A3 respectively, resulting in the agents A1 and A3 connecting as shown in Figure 17E.

An example of the handoff mechanism handing from one component to another will now be described with reference to Figure18. In particular, in this example, the schematic includes three components A, B, C, each of which has respective agents A1; B1, B2; C1, C2.

In this example, the agent A1 gets the address of the agent B1 from the schematic purchase order. Similarly the agent B1 gets the address of the agent A1 from its schematic purchase order. Using the agent connection mechanism outlined above agents A1 and B1 connect and authenticate.

Agents A1 and B1 perform their information transfer, and this results in the agent B1 deciding to hand off the agent A1 onto the agent C1. The component B obtains the agent address of the agent C1 by having the agent B2 communicate with the agent C2. The agent B1 then uses its authenticated link to the component A, and sends a hand-off request together with the agent address of the agent C1.

The agent A1 simply disconnects from the agent B1 and connects with the agent C1. Should the agent C1 be busy with a connection elsewhere. The agent A1 simply waits for the agent C1 to become available. Similarly the component B obtains the address of the agent A1 and transfers this to the agent C1, allowing the agent C1 to reconnect to the agent A1.

Bundling

Often a component will require a number of agents to resolve information for a specific task. Since these agents are related it makes sense to group the agents into a bundle to hide the complexity and deal with the bundle like a single agent. This greatly simplifies the schematic and reduces errors.

Thus, the purpose of the bundling is to manage agents more effectively. Although not strictly necessary bundling allows related agents to be attached to each other so that its relationship is preserved making the management of large numbers of agents an easier task.

Special components provide the service of bundlers/debundling and these will hereinafter be referred to generally as Bundlers. In this example, bundlers have three agents - two "inputs" and an "output", whereas bundlers operating to debundle (which may be referred to as "debundlers") have two "outputs" and an "input". The terms output and input are inverted commas as the bundler and debundler perform almost exactly the same task.

They both gather the addresses of a pair of agents and send it through a third agent, however the bundler is the one that initiates the communication, and so gathers the addresses first - making the pair of agents inputs and the lone agent an output. As soon as the debundler has received the addresses through its input the roles are reversed. Once the bundler and debundler have swapped the information, they handoff the connected components to each other. If either of these connected components is a bundler or debundler, the process begins again.

Figure 19A represents two agents A1, B1 from respective components A, B coupled through a bundle component BUNDLE, which provides bundle agents BU1, BU2, BU3. The bundle agent BU3 is used to connect to the component X. The bundle agents BU1, BU2, BU3 are indistinguishable from a normal agent.

In use, the component BUNDLE depicted in Figure 19A receives connections from the agents A1, B1 and presents the agent BU3. The role of the agent BU3 is to provide the addresses of the agents A1, B1 to the component X.

In the example shown in Figure 19A, the components A, B, X receive addresses of the agents BU1, BU2 and BU3 respectively from the schematic purchase order. Similarly the bundle component BUNDLE gets the agent addresses A1, B1 and X1 from a respective schematic purchase order. The agents A1, BU1 connect and authenticate while the agents B1, BU2; and, X1, BU3 do the same. The component X negotiates with the component BUNDLE and determines that the payload of the agent BU 3 represents a bundle.

Accordingly, by using the hand-off mechanism as described above, the component X can determine the addresses of the agents A1, B1, and order the bundle component BUNDLE to hand-off A1, B1 as shown in Figure 19B. Thus, in Figure 19B the component X

through agent BU3 learns of the agents A1, B1. The component X then orders the bundle component BUNDLE via the agent BU3 to hand-off the agents A1, B1 to the agent X1 and X2 respectively. The bundle component and its respective agents has then completed it's service and can retire.

The bundling component BUNDLE as shown in Figure 20 has no concern as to the nature of the agents A1, B1. As far as the bundling component is concerned, they are any two agents and their payload is irrelevant. This enables cascading of bundling components as shown in Figure 20. Any number of bundling components may be cascaded.

In addition to providing bundle components for bundling agents, debundling components are provided for performing the opposite function.

An example of this is shown in Figure 21, in which a component A is coupled to a debundling component DEBUNDLE, which in turn is connected to two components X, Y, as shown. In this example, the debundling process starts with the component agents A1, DBU1; DBU3, X1; and DBU2, Y1 connecting and authenticating as specified in the purchase order.

The debundling component then learns the addresses of the two agents represented by the bundle. It then requests X1 and Y1 to hand off in accordance with the addresses provided by agent A1. At this point the debundling agent has then completed its service and can retire.

An example of the manner in which bundles of agents can be used will now be described with reference to the example shown in Figure 22A.

Internal to a Component P, a Bundler 1 is connected to a Component A and a Component B. The output of the Bundler 1 is connected to one of the inputs of a Bundler 2. The other input is connected to a Component C. The output of the Bundler 2 is the bundle of agents that will be connected to an external agent of the Component P.

The Component P is connected to another Component Q, and the bundled agent is, internal to the partner component, attached to a Debundler 1. The Debundler 1 is attached to a Component D and a Debundler 2. The Debundler 2 is attached to a Component E and a Component F.

During automatic handoff, the Bundler 2 will be connected to the Debundler 1. The bundler 2 obtains the address of the agent connected to the Component C and the Bundler 1 respectively and sends these to the Debundler 1, which will respond with the address of the agent connected to the Component D and the Debundler 2 respectively. The Bundler 2 and the Debundler 1 will then perform a handoff of this respectively agents so that the Component C will be connected to the Component D, and the Bundler 1 will be connected to the Debundler 2 as shown in Figure 22B. The Bundler 2 and Debundler 1 are now finished, and retire.

Now the Bundler 1 will send the addresses of the agents connected to the Component A and the Component B to the Debundler 2, which will respond with the addresses of the agents connected to the Component E and the Component F. The Bundler 1 and the Debundler 2 will then perform a handoff so that the Component B will be connected to the Component E and the Component A will be connected to the Component F. The Bundler 1 and the Debundler 2 are now finished, and retire as shown in Figure 22C.

Some very complex patterns of bundling and debundling can be used provided that the patterns are symmetrical around the link going between the bundlers and the debundlers.

These patterns can span across multiple components and multiple levels of the component hierarchy symbolised in Figure 22D.

Bundle Arithmetic

IN this example, there are just two primitive components used in agent arithmetic. They are the:

- Bundle Component.
- Debundle Component.

In this section two examples of bundle arithmetic are given. These are:

- Bundle Reordering
- Rake-out

With components bundle and debundle a number of useful operations can be performed on bundles. An example of agent reordering can be found in Figure 23. In this example, the component U represents a debundle (unbundle) component, with the component B representing a bundle component. Accordingly, in this example, the agents a, b are reordered within the bundle by extracting the agents a, b from the bundle and reassembling the bundle with the agents b, a in the reverse order.

Rake-out is when an agent is extracted from a particular agent bundle. An example of agent rake-out can be found in Figure 24. In this example, a respective agent c in a bundle of agents a, b, c, d is required to be separated from the bundle. This is achieved by debundling the bundle using the debundle components U until the agent of interest is available, then rebundling the bundle using the bundling components B.

Constructor

The constructor is a software application provided at the entity stations 5 to allow the entity stations to implement components. In particular, the constructor is adapted to receive a purchase order generated by the end station 3, or any portion of the forum, and then use the purchase order to create a new component instance, together with appropriate agents.

Once this is completed, the local constructor activates the agents associated with the component, in a process hereinafter referred to as presenting the agent. All agents by definition are connected to another agent associated with another remote component . When an agent is presented, it is made available for interaction with its counterpart operating in its remote constructor.

An example will now be described with reference to Figure 25, which represents a component CP prior to release to the constructor. In particular, the component CP is formed from three sub-components A, B and C, and is provided with four agents W, X, Y and Z.

Upon receiving the purchase order from the end station 3, constructor will initiate the construction of the component instance that is to perform the service in the respective build process.

In order to achieve this the constructor operates to:

- Submit purchase orders to the entities supplying the services associated with the respective components A, B and C, as shown in Figure 26;
- Provide the data required to each of the sub-components A, B and C, to allow each of the sub-components to perform the services defined therein; and,

- Supply sub-component agent addresses to allow the components A, B, and C to be erected once their agents have terminated.

In general, the constructor contains many components at various stages of erection but for the purposes of the following explanation the constructor will only operate on one component.

In particular, the constructor presents the agents W, X, Y, Z to allow these to communicate with agents of other components, as required by the schematic. In addition to this as the component CP has sub components A, B and C, a number of internal interactions must also be resolved.

In general, users of the component CP are unaware that the component CP is formed from a number of sub-components A, B, C. Accordingly, the user needs not provide details of the interactions that need to be performed between the components A, B, C as this will be determined by the entity providing the component CP. Accordingly, when the constructor orders the components A, B and C, the constructor will also provide details of the interactions required between the components A, B, C.

The constructor also presents temporary agents T1, T2, T3 and T4 as shown in Figure 27 to provide the interface between the internals agents of the component CP and the sub-components A, B and C. These temporary internal agents T1, T2, T3 and T4 are presented along with the external agents W, X, Y, Z as shown in Figure 27, thereby allowing the component to be implemented with all the agents having a chance to resolve and connect.

Construction Site

The construction site is the location at which the final executable code is erected.

In operation, the construction site is typically provided at the end station 3, or a processing system within the forum, with access to the construction site being strictly controlled so that only the necessary portions of the site are exported to the relevant entity. However, it will be appreciated that this is not essential if other architectures to that shown above are used.

In any event, each entity working on a build will eventually interact with the construction site to erect their portion of the code. The site starts out as a linked list with zero elements. As each sub-contract is let the linkage is exported to the contractor and a number of services can be performed on the list. These services are:

- Add Element - This is used to create sites where bytes will be deposited. It requires supplying Cargo, Component ID and Position.
- Delete Element - This is used to destroy an unused element and requires Component ID and Position.
- Construction Complete - Requires supplying Component ID.

**Specific Examples**

Alternative techniques are outlined in a number of specific examples set out in detail below.

First Specific Example

The first example results in the construction of a schematic containing a single component. This straightforward example serves to highlight the steps involved in construction.

Figure 28 shows an external component representation Add1 of a component "Add1" that offers an "add" service. An internal component representation is shown in Figure 29. As

shown the component include two inputs IN1, IN2, and two outputs OUT, EXCEPTIONS.

In this example, the component representation Add1 is layered-up into a schematic "Add1 test" as shown in Figure 30. This schematic consists of an outer box called the root component, the component representation Add1, and connected to this are four internal agents IN1, IN2, OUT, EXCEPTIONS denoted by the crosshatched boxes of Figure 30. In this example, all agents are connected in pairs with a single line.

Every build begins with the starting of a root component server, which in this example corresponds to one of the processing systems outlined in the example above. Theoretically any one of the base station 1, the end stations 3 and the entity stations 5 in the example set out above could act as the root component server, although in this example it is the end station 3 of the user.

When the root component server receives a build request it proceeds to construct the root component. In this example the root schematic associated with the root component is loaded and scanned. All component representations in the root schematic are then identified and secondary build requests are issued by the root component server. In this case only a single build request is issued to an "Add1" component server since the example schematic only contains this one component.

On the "Add1" component server, which may for example be one of the entity stations 5 operated by a respective entity and positioned at a remote location, there is an "Add1" schematic associated with the component in much the same way as the root schematic is associated with the root component. Figure 29 represents the "Add1" schematic residing on the "Add1" component server. Of note is that the schematic contains nothing. Accordingly, the component Add1 does not include any sub-components.

When the "Add1" component server receives the secondary build request that was issued from the root server it creates a new instance to serve as a vehicle for providing the "add" service to this new customer. Similar to the actions of the root component server the "Add1" component server loads the schematic associated with "Add1" and scans it for any component representations so that further build requests can be sent out. In this example however the "Add1" schematic is empty and thus no further build requests are issued.

It is important to note that the "Add1" build request issued by the root component server supplies information about the agents associated with the "Add1" component. That will allow the "Add1" component instance's agents to connect to the correct destination, which in this case are on the root component server.

The next step for the root component server is to scan its root schematic for internal or external agents. Since it is a root component it will not contain any external agents. However the schematic does include the four internal agents IN1, IN2, OUT, EXCEPTIONS as shown in Figure 30.

The root component server then proceeds to create these internal agents and they begin to contact their respective partners, namely the agents IN1, IN2, OUT, EXCEPTIONS on the "Add1" component server. The "Add1" component server performs a similar function. The "Add1" schematic is scanned for agents and only four external agents IN1, IN2, OUT, EXCEPTIONS are found. The "Add1" component server then proceeds to create these external agents and they also begin to contact their partners at the root component server.

Since the root component server and the "Add1" component servers were successful in creating their internal and external agents respectively, they connect successfully.

At this point the root component server has established four communications paths to the "Add1" component server and the user can now utilise the service at the "Add1" component server via the agents available on the root component server.

Since the "Add1" schematic is empty the service supplied by the entity must be performed manually by an operator stationed at the "Add1" component server. Thus when the user of the root component server enters a number in agents IN1, IN2, respectively, the operator at the "Add1" component server will receive these numbers and can then perform their service. In this example the operator at the "Add1" component server must add the numbers manually and return their result using the agent OUT.

If the "Add1" component wished to verify the formatting of the inputs or outputs, for instance, the base of the number system to be used, or the number of decimal places to be used, the agents are free to send additional payload packets until all parties are satisfied. In this case, as there is no need to transfer further information, the agents are free to terminate the link.

Once the transaction is compete the agents can be terminated at both the root component server and the "Add1" component server, and the root component instance and "Add1" component instance can be retired respectively.

Although this example is quite simple it serves to highlight the concept of the component servers, how the agents interact and that a service performed remotely at the component server, which will typically be situated remotely to the end station 3. Thus, for example, the root component server may be located in Townsville, with the "Add1" component

server in Rome. Despite this, the location of the "Add1" component server is transparent to the user in Townsville.

The last point to note is the use of the EXCEPTIONS agent, which can be used should the entity providing the Add service have any difficulty in performing the service as contracted, the exception agent can be used to communicate the difficulty.

Second Specific Example

The second example is substantially the same as the first example, except that in this example, the provision of the "Add" service is through the use of a component "Add2" which is automated. An example of the component representation Add2 of the component "Add2" is shown in Figure 31. As shown the component representation Add2 is substantially the same as the component representation Add1.

This example shows it is very easy to automate the services of simple components. In particular, automation is achieved by monitoring states of the agents belonging to each instance of an automated add, and performing certain tasks as the states change.

In this example, at the time when the agents IN1, IN2 of the "Add2" component have received payloads and the agent OUT is connected, the payloads from the agents IN1, IN2 are added together and sent out as a payload packet via the agent OUT.

As this automated add component is designed to work for only decimal addition, it will send a message indicating that an input was incorrectly formatted if an input is not a valid decimal number. When the agent OUT has received a termination indication, the agents IN1, IN2 send terminate packets to their partners.

The result of this is that if the root component sends the numbers 5 and 4 to the automated “Add2” component, the result, 9, will quickly arrive at the agent OUT of the root component. If the root component was to then send the number 6 via the agent that had sent the number 5, the number 10 will arrive at the output, allowing for any corrections without the need for a restart.

If the root component were to send the letters "five" and the number 4 via the agents IN1, IN2 respectively, agent IN1, will receive the message "Number formatted incorrectly" from the automated “Add2” component.

Third Specific Example

The third specific example extends the complexity of the *add* component allowing it to handle input numbers of different number bases, such as decimal and hexadecimal. This is an addition that can accept inputs as either decimal or hexadecimal numbers and can output either a decimal or hexadecimal number.

Figure 32 is an example of the component representation Add3 of an “Add3” component. The “Add3” component includes inputs and output having agents IN1, IN2, FormatIN1, FormatIN2, FormatOUT, EXCEPTIONS, as shown. In this example, the “Add3” component is a compound component containing many internal components that have been selected and arranged in such a way as to perform this more complex service. Figure 33 is the internal schematic of the “Add3” component and this introduces a number of new components in order to perform this more complex service.

Detailed in the schematic shown in Figure 33 are a number of “convert” components, which perform the service of converting a number in a particular format to another format. In this way a user of the “Add3” component can specify the format of each number.

Because the automated "Add2" component will only work with decimal numbers it is rather fragile. However the "Add3" component performs the same basic service but is much more robust in that it can handle input numbers of many formats and even produces the output number in any format desired. This is despite using the fragile "Add2" component.

Formatting is achieved using the agents FormatIN1, FormatIN2, FormatOUT.

Each of the input numbers supplied to the agents FormatIN1, FormatIN2, are converted to decimal using the respective "convert" components, before being transferred to the "Add2" component. Each "convert" component has four agents IN, FormatIN, OUT, FormatOUT.

The number from the respective IN agent of the "Add3" component is presented at the agent IN, and is interpreted as being of the format specified by the respective FormatIN agent of the "Add3" component.

In this example, three instances of the "convert" component are used, one for each of the agents belonging to the "Add2" component. This is necessary as the "Add2" component only works with decimal numbers.

When the "Add3" component is used in a build all of the external agents IN1, IN2, FormatIN1, FormatIN2, FormatOUT, EXCEPTIONS, will become connected to their respective partner agents on the root component (not shown).

The "Add3" component will then build its internal schematic corresponding to Figure 33. This will send build requests and agent connection details to all the sub-components. All

the external agents IN1, IN2, FormatIN1, FormatIN2, FormatOUT, EXCEPTIONS, of the "Add3" component are connected directly to the respective "convert" sub-components and so they are handed off resulting in the subcontractor "convert" components connecting directly to the partner agents on the external agents. Apart from the external agents the "Add3" component has three internal agents DECIMAL, which are set to automatically deliver their payload as soon as the agent has connected. As these agents send their payload automatically and send terminate as soon as they receive a terminate packet, they are in effect automated agents.

If the conversion component is automated, then the entire "Add3" component is in effect automated. This is a good example of the building of complex programming components from simpler programming components, and also of non-primitive automation. The "Add2" component was automated, but as there is no way of performing the task via a sub-schematic it is classified as a primitive component. In effect, all software produced with these techniques will be derived out of primitives at the lowest level.

The "Add3" component is more versatile than the automated "Add2" component. The purchaser of an "Add3" component will be able to perform addition without worrying about how the conversions work, demonstrating a level of complexity hiding.

<u>Fourth Specific Example</u>

This specific example is the most complicated add component to be discussed.

In this example, an "Add5" component also provides additional testing and functionality to the "add" service. However it will be noted that the discussion only adds sufficient complexity to illustrate certain key features of the technology. The "Add5" component demonstrates the use of agent bundling, basic input testing, schematic selection and exception handling.

The "Add5" component extends the "Add3" component by including input validation for the Format specification and uses bundling to reduce the number of external agents. Figure 34 shows the component representation of the "Add5" component. It has 5 agents that are briefly described in the table 1 below:

Table 1

| Agent | Description |
|---|---|
| IN1 | Input number 1 (Bundle of number value and number format) |
| IN2 | Input number 2 (Bundle of number value and number format) |
| FORMAT | Specifies the required format of the output number (i.e. decimal or hexadecimal) |
| OUT | Output result of the addition (Bundle of number value and number format) |
| EXCEPTIONS | Indicates if any build exception occurs. |

Figure 35 shows the internal schematic of the "Add5" component.

This example introduces 6 new components (Dup, Validate String, Or, U, B, and a schematic selection component C11), which are briefly described in Table 2.

*Dup*

This component duplicates the payload of its IN agent to each of its output agents. This component is used whenever a piece of information is needed multiple times.

*Validate String*

The Validate String component works by comparing the input string (In) to a list of valid strings on the LIST agent. In this example the list would be "Decimal" and

"Hexadecimal". If the input string is valid than it is passed as being valid resulting in the OUT(checked) agent returning "true". If the string were not valid the OUT(checked) agent would return "false". An additional service of the validate string is to reformat the input string into a specified output format. This might include the removal of white space (space, and carriage return character), converting the string to lower case and also trimming the string length. The FORMAT agent specifies this output formatting.

Table 2

| **Component** | **Designators** | **Brief description** |
|---|---|---|
| Dup | C4, C3, C9 | Duplicates the payload of its IN agents to all of its OUT agents. |
| Validate String | C5, C6, C7 | Checks an input string against a list of valid strings. It outputs the result of the checking and reformats of the input string to a given specification. |
| Or | C8 | OR Boolean logic, output is true if any input is true, else output is false. |
| U | C1, C2 | Unbundle. Splits a bundle of agents. |
| B | C10 | Bundle. Combines two agents into a bundle. |
| Schematic selection | C11 | Selectively builds one of *n* schematics based on the payload of the selector agent. |
| Add4 | | Same as an Add3 except that the service is performed manually, allowing much more power resolution of problems than the automatic Add3 component. |

*Or*

The Or component is used when ever it is desired to trigger an event from multiple sources. It combines all its inputs so that if any of them are true it will output a true.

When Or is first purchased all of its input agents are undefined as each of the agents will only have just connected and will thus have no payload. The Or component then waits for agent payloads until it has sufficient information to decide what output it should deliver. It will output a "true" as soon as any of the input agents have a payload of "true", otherwise it will wait until all input agents are specified as "false", at which point it will output "false".

*B (bundle), U (unbundle)*

The Bundle component bundles agents together, whilst the Unbundle component splits a bundle of agents apart. The bundle/unbundle process is described in more detail in the bundling section of the patent.

*Bundled Agents*

The agents IN1, IN2, OUT use bundling to combine two agents together, one representing the value of the number and one for specifying the format of the number. This bundling assumes that the number format is in the first position of a bundle, and the number value is in the second location. This combination of these two particular agents will be referred to as a number bundle. To ensure compatibility the agents IN1, IN2 must only be connected to agents that produce a number bundle and the OUT agent must only be connected to an agent that can handle a number bundle. For more information about bundling refer to the section on bundling and unbundling.

The number inputs In1, In2 of the agents IN1, IN2 of the "Add5" component are unbundled by the components C1, C2 to get the number value and number format agents. After the components C2, C1 perform their service the number value of In1 becomes connected to the agent A37 and the number format becomes connected to the agent A26.

For the number input In2 the number value becomes connected to the agent A39 and the number format becomes connected to the agent A29. The number formats for each of the inputs In1, In2 is then verified and formatted by the Validate String components C6, C7. The agent A19 specifying the output format is also checked using C5.

*Duplicates and Validate Lists and Formats*

The Validate String components allow the inputs to checked and formatted based on predetermined requirements.

In this example, it is necessary want to ensure that the Format specified for the Add component used in the addition component C11 is compliant with the agents FormatIN1, FormatIN2 and FormatOUT. Each of the Validate String components C5, C6, C7 require a specified list of valid strings, which is supplied by the internal agent A14, and duplicated three times by the Dup component C4. In this case the payload for the internal agent A14 would be "Decimal, Hexadecimal". The internal agent A10 specifies the required output format for the three Validate String components, which maybe something like "No white space, lower case". This is achieved by duplicating the string received from the internal agent A10 using the Dup component C3.

*Selective Construction*

The OUT(checked) agents A23, A53, A31 of the Validate String components indicate whether each of the Format specifications passed the input verification. It any of them fail, additional functionality is required to be performed manually. The OUT(checked) agents A23, A53, A31 are combined into a single agent A36 using an Or component C8, which will have an agent payload specifying whether to use a manual or automatic add provided by "Add4" or "Add3" respectively.

The core part of the schematic in Figure 35 is performed by the addition component C11. This component is special in that the sub-schematic of the addition component is dependent on the payload of a selector agent A36.

Internally to the component C11, the agent A36 selects whether to construct an automated "Add3" or manual "Add4" component. If the inputs In 1, In 2 and Format provided to the agents A1, A3, A19 respectively pass the input checking performed by the validate string components C5, C6 and C7 then an "Add3" component is used to perform the addition operation. Conversely, if the inputs don't pass the validation than the addition is performed by a manual "Add4" component, allowing the inputs to be studied and queried by a human. If the information makes no sense whatsoever, an exception will be generated and passed to the parent component.

The addition component C11 demonstrates the use of selective construction, where a different sub-schematic is built based on the information provided by an agent. Although only a simple case has been presented in this example, in general this technique is extremely powerful. It allows the schematic design to be selected based on the input information to the component. It also allows the construction to be completed in stages. As each stage completes it can trigger the construction of the next stage.

Staged construction can greatly minimise the total number of component instances used at any one time, minimising computer resources. It also allows a trade off between a completely serial construction (i.e. the components are purchased and built one at a time) and a completely parallel construction (i.e. all components are purchases and built at the same time). A serial construction minimises computer resources but is inherently slower than a parallel construction.

The addition component C11 only has a single component in its sub-schematic, however in general it can contain a schematic of any size. The operation of the addition component C11 is transparent as for as the remainder of the schematic is concerned, and in this example acts just like a normal component.

In fact it could be implemented using a component server that selects the sub-schematic based on an agent payload. Alternatively it could be implemented directly as part of the "Add5" component. In this case each of the agents A36-A43of the addition component C11 are implemented as internal agents, which get handed off when the sub-schematic of the addition component C11 is built.

*Component Output*

The output of the addition performed by the addition component C11 is combined into a number bundle using the bundle component C10.

Fifth Specific Example

In the previous four specific examples, it has been demonstrated how it is possible to perform processing of agent information. None of the previous examples have dealt with the construction of code, but were instead computer programs performing some task, which in these examples was the calculation of an addition. It will be appreciated from this that although the examples set out in the flow charts of Figures 1, 3A to 3B, and 10A to 10E are examples of producing software, these could also be used in performing processing operations directly.

In any event, the fifth specific example extends the concept by using the technology to construct executable code directly byte by byte.

In this example a component is outlined that constructs code, that when executed on an IBM PC will display a pixel on the screen, given a position and a colour. The code produced is not however a complete program, but instead a code fragment constructed to the requirements of the program in which this component has been designed into. This component produces x86 machine code as its output. The details of this component are specific to IBM PC architecture, and so the details will only be briefly described.

Figure 36 shows the component representation Put Pixel of the "Put Pixel" component. It has five input agents X, Y, Width, Colour and Screen and two output agents Exceptions and Build.

A description of each of the agent is shown in Table 3 below.

Table 3

| Agent | Input/Output | Description |
|---|---|---|
| X | Input | Horizontal position, from left to right in pixels, to draw the pixel. |
| Y | Input | Vertical position, from top to bottom in pixels, to draw the pixel. |
| Width | Input | Width of the screen in pixels. This must match the current mode of the screen and must be set up previously in the program. |
| Colour | Input | Colour of the pixel (colour code 1 byte) |
| Screen | Input | Segment address of the screen (Typically 0xA000) |
| Build | Output | Code produced by the component |
| Exception | Output | Indicates any build exceptions detected by the component. |

Figure 37 shows the internal schematic for the "Put Pixel" component. This component uses the "Add5" component from before, plus a number of new components (Mul, Mov, Seg Prefix, and Build).

An outline of the components is given in table 4 below.

All of the pixels on the screen are stored in video memory - which is just a certain set of addresses in normal memory. Changing anything in those areas of memory results in a change on the screen. Memory for IBM PCs is referenced by segment and offset. The segment selects large regions of memory, while the offset allows access to all the locations with in a segment. The screen memory is located at the segment address A000 hexadecimal. The offset determines the position of the pixel, and the value stored at that offset determines the colour of the pixel.

Table 4

| **Component** | **Description** |
|---|---|
| Mul | Similar to the Add5 except that it performs a multiply. |
| MOV | x86 Move assembly instruction. This creates machine code for move. |
| Seg Prefix | Calculates the prefix for the next move instruction based on which segment register is specified. For example: x86 segment register *es* corresponds to 38 in hexadecimal. |
| Build | Concatenates the code produced by two components. |

The offset of any point (x, y) on the screen is given by (y * width) + x as the screen image is store as a linear array, one row at a time. The multiply (C1) and add (C2) perform this calculation.

The "Put Pixel" component assumes that the screen has been set to the correct video mode, and that the segment register specified by the screen agent is set to A000 hexadecimal.

The "Put Pixel" component creates three MOV assembly instructions. The pixel colour is moved in to the AL register with C3 and C6. The calculated pixel offset (A14) is then moved into the DI register by C4 and C7. Then finally the value in the register containing the colour (AL) is moved to the memory location described by the value in the segment register and the value in the register containing the offset (AL -> Segment:DI).

The assembly instructions use the MOV component, which takes a source and destination, and outputs the appropriate hexadecimal machine code for the requested MOV instruction.

The code produced by the individual components is collected and combined by the build components producing the deliverable executable code out the Build agent (A48).

Sixth Specific Example

The final specific example demonstrates the usage of the Put Pixel component of Figure 36 to create stand-alone program as detailed in Figure 38.

The "Put Pixel" component is used in conjunction with a "Setup Screen" component - responsible for changing the video mode and setting a segment register to the value of the screen memory. The address of the screen memory is input to the "Put Pixel" component, and the outputs are the bytes corresponding to the code produced, and the segment register used to store the memory, which is used by the put pixel component. The program is to be booted from a floppy drive, and will clear the screen (from the setup screen component) and then display a single pixel at location 20,50 of colour red.

The put pixel component gets its inputs from number bundles - bundles containing numbers and the corresponding formats of the numbers - which are input to the agents x, y, and width. The colour input is input directly from an internal agent of the root schematic. As mentioned above, the segment register used to store the segment address of video memory is input from the setup screen component. The output of the Put Pixel component, **build,** delivers the bytes generated by the component to a build component which appends the bytes to the bytes produced by the setup screen component. The Build component then delivers the complete program to a Boot component, which is responsible for correctly formatting the bytes as required for a boot disk.

**Summary**

Accordingly, the above described system allows users to perform data manipulation or generate computer executable code by defining combinations of components. In this case, each component corresponds to a respective data manipulation service and accordingly, the component combination defines a sequence of data manipulations which when performed will result in the desired data manipulation being performed or the desired executable code being generated.

The components are generally provided by respective entities which is capable of performing the data manipulation service defined therein, and this may be achieved either manually or in through automated procedures. Accordingly, in order to allow a user to define a suitable component combination, the components are usually made available through a centralised system, which is often referred to as a forum.

In order to allow the data manipulations to be performed, it is necessary to be able to define the component combination with sufficient detail to allow the components to interact. In order to achieve this, in the examples described above a schematic is defined

which sets out the components to be used, and the interactions therebetween. The schematic is typically defined using a suitable GUI, which therefore allows users to select components presented on the forum, drag and drop these into the schematic, and define suitable connections between the components to define the component interactions.

Once the schematic is completed, this may then be implemented in a process known as a build.

During a build a respective component instance will be generated for each component in the schematic. Each component instance will be created on a respective component server which is typically implemented using a processing system provided by the respective entity. In use, when the system is implemented data is transferred between the respective component instances with each component instance performing required data manipulations on received data, so as to provide an output of manipulated data as required.

It will be appreciated that the component instances must be capable of communicating with each other, and in particular, must be capable of transferring information and data in a form that can be understood by both components.

In order to achieve this, communication between the components is performed using agents, with a respective agent being provided for each component input and output. Thus, an agent associated with an input on one component will cooperate with an agent associated with an output on another component.

The interaction between the agents occurs in two stages including:

- Negotiation; and
- Data transfer.

Accordingly, the agents will first negotiate with each other to determine a common data format which can be used to transfer data between the respective components, before proceeding with the data transfer as required. Thus, the agents represent the only form of interaction between the components.

When a schematic is to be built during a build process, purchase orders are sent to each entity providing components within the schematic. Each entity will then construct a respective component server including a respective component instance together with any associated agents. Once this has been completed, the agents perform any required negotiations before the transfer of data between the components occurs in order to allow the components to perform the respective data manipulations embodied by the component.

Thus, when an entity receives a purchase order for a respective component this will specify connections that need to be formed between agents associated with the component, and other agents. In particular, this will include the agent addresses of the other agents so that when the component instance and corresponding agents are constructed, the agents will be able to communicate directly with the other agents.

Both the negotiation and data transfer will involve the transfer of data packets including a header and associated payload. It will be appreciated that in the case of negotiations the payload will typically include a list of acceptable data formats that may be handled by the agent, or the like. Thus, a first agent will transfer a suitable list to a second agent with the second agent responding with an indication of a format which is acceptable. In the case of transferring data to be manipulated, the data will be included in the payload.

A number of additional features associated with agents will now be described.

Combining Agents

As described above, each agent interacts with one other agent, which is typically associated with another component. In general, components may include many inputs and outputs and therefore may have many agents. If it were necessary for individuals to define connections between each agent of each component when creating the schematic, this would be an extremely onerous task.

Accordingly, in order for this process to be more efficient it is usual for agents to be combined. Thus, in the example described above with respect to Figures 22 it can be seen that each of the components P, Q include respective sub-components A, B, C, D, E, F, which in this example each have a respective agent. It is possible for each of these agents, to be provided with corresponding agents on the respective component P, Q, which would therefore require three connections to be formed.

Accordingly, it is typical for related agents to be combined, thereby allowing a single connection to be defined, as shown. This may be achieved using complex payloads and/or bundling.

Complex payloads are formed when the payloads from each of the agents are combined into a single payload. In this case, the component P could have a single external agent, which provides a single payload which corresponds to a concatenation or combination of each of the payloads of the agents of components A, B, C. However, in order for the components D, E, F to respond, it is necessary for the complex payload to be deconstructed by the component Q, to allow respective individual payloads to be formed, which can then be provided to the agents of the components D, E, F. This can be a time consuming and computationally expensive process.

In the case of bundling, agents are combined via the use of a bundle component such that two or more agents are effectively treated as a single agent. An unbundling component is then used to deconstruct component bundles as required. This in turn allows agent hand-off to be implemented so that agents not actually involved in any interaction can transfer the interaction requirements to other agents as described above.

This allows complex interactivity to be demonstrated between multiple components whilst providing a simple mechanism for this to occur.

Hierarchical Bundling

It is typical for agent bundles to include a large number of agents, up to for example a hundred or more. In order to improve the efficiency of the bundling/debundling process, it is typical for agent bundles to be arranged hierarchically so that those agents or bundles of agents which need to be accessed on a large number of occasions are more easily accessible.

A number of different hierarchy structures may be used, such as linear, or dynamic hierarchies. However typically bundles are arranged in a hierarchical tree fashion as shown for example in Figure 39.

This presents the structure of an example bundle. In particular, in this example, the bundle B contains the agents $A_1$ - $A_{12}$ arranged as shown. Thus some of the agents $A_1$ - $A_{12}$ are arranged with bundles $B_1$ - $B_8$ which are themselves contained within the bundle B. Thus, if a component requires interaction with the agent $A_1$, the bundle may be debundled at a first level to provide access to the agent $A_1$. In this case a debundle component is used to break the bundle B down at the first hierarchical level and extract the agent $A_1$ allowing this to be provided to a respective agent as required. At this time the bundles $B_1$ - $B_2$ are also typically extracted from the bundle B. An example of this is

shown in Figure 40.

In this example, a component 1000 is provided having sub-components 1001, 1002 and 1003 as shown. In use, the component 1002 is adapted to operate on a payload provided by the agent $A_1$ in the bundle B. Accordingly, in use the component 1000 operates to receive the bundle B at the agent 1004 which operates to transfer the bundle to the agent 1005. This will typically be achieved by a hand off mechanism with the agent 1004 simply handing off the bundle B to agent 1005.

In any event the component 1001 is a debundle component which operates to debundle the bundle B to the first level in the hierarchy. The agent $A_1$ is then output via the agent 1006 with the bundles $B_1$, $B_2$ being output via the agents 1007 and 1008 respectively. The agent $A_1$ is transferred to the agent 1009 allowing the component 1002 to obtain the payload of the agent $A_1$ and provide any data manipulation as required. An output may then be provided via the agent 1010.

It will be appreciated that this is all that is required in order to interact with a particular agent within a bundle. Thus, the bundles $B_1$ and $B_2$ may themselves be transferred on to other components for further processing. Similarly, the agent $A_1$ may now have fulfilled its purpose. Whether further use is made of the bundle, or any agents extracted therefrom is not essential to the bundling process.

However, in this example, for illustrative purposes only, the component 1000 is adapted to provide a modified bundle B' at the output agent 1015.

Thus, in this example, the output provided at the agent 1010 may include a modified version of the payload from the agent $A_1$, as indicated at $A_1'$. The agent $A_1'$, together with the bundles $B_1$, $B_2$ are then transferred to the component 1003 via the agent 1011,

1012, 1013 respectively. The component 1003 operates to rebundle the bundle B as now modified, indicated by B' providing this via the agent 1014 to the output agent 1015, as shown.

Thus, the component 1000 allows the payload of the Agent $A_1$ to be modified. It will be appreciated that the rebundling of the modified agent $A_1'$ into the bundle B' is not required, and instead, agents may simply be extracted from bundles and used as required.

Figure 41 shows a modification of the component 1000 in which an additional agent 1020 is provide additional interactivity with the payload of the agent $A_2$. In order to achieve this an additional debundle component 1021 and bundle component 1022 are used as shown.

It will be appreciated that the functionality of this is similar to that described above and this will not therefore be described in any further detail.

Chaining

A further useful technique in implementing the build process is a technique known as chaining.

In particular chaining operates by transferring agents, or more usually agent bundles, through a schematic allowing the agent or agent bundles to have payloads modified as required, in a manner similar to that described above. In addition to this, the chain is intended to pass through the schematic, or a portion thereof unbroken. This allows modification of payloads to be passed through the system, and returned as required.

This may be used for example to allow data to be output to a predetermined location as specified. An example of this will now be described with reference to Figure 37 which is

the example of a putpixel component.

In particular, this example, as previously described, operates to construct applications software for positioning a pixel on a display. In this case, the putpixel component is adapted to generate executable code which is provided via a build agent A48 as shown. In general, the executable code will, when executed by a suitable processing system, cause the processing system to position a pixel at a required location on a screen.

The executable code will need to be constructed at a specific memory location on a root server, and the agents in the schematic will therefore need to know the memory location at which the executable code is to be constructed.

In order to achieve this, a build bundle formed from a bundle of appropriate agents required to construct the executable code, is provided by the root server. This will typically be achieved by having the root server implement appropriate agents as required. In this case, the payloads of agents in the build bundle will include details of the required memory locations. Thus the build bundle will specify memory locations at which specific data is to be constructed.

The root server transfers the build bundle to the build agent A48 and this is in turn passed on to the build output agent A35 of the build component C12. The bundle is then transferred back via the agents A34, A33, A32, A31, A30 to the agent A26. This allows the move component C6 to generate a move assembly instruction and include this as a payload within the build bundle as required. The move assembly instruction which is provided in the build bundle will be associated with a respective memory location as defined in the build bundle by the root server.

The build bundle, having been modified in this manner, is then transferred back via the

agents A26, A30 to the agent A29 of the move component C7. The move component C7 will generate a corresponding move instruction and include this in the build bundle, allowing the build bundle to be transferred back via the agents A31, A32, A37 to the agent A36. In this case, the seg prefix component C8 will calculate a prefix for a next move instruction and include this in the build bundle as a respective payload before transferring the build bundle back via the agents A37, A33, A34, A39 to the agent A38. The move component C9 will then insert a further instruction before providing the build bundle back to the build agent A48 and hence back to the root server.

In the above example, the path of the build bundle through the putpixel component passes through the agents A48-A35-A34-A33-A32-A31-A30-A26-A30-A29-A31-A32-A37-A36-A37-A33-A34-A39-A38-A39-A35-A48 to form a chain.

Accordingly, it will be appreciated that in this instance the put pixel component is adapted not only to receive inputs at the agents X,Y, WIDTH, as required but also to receive a build bundle via the build output A48. In this case, as soon as data is received at a respective one of the inputs X,Y, WIDTH, this will be transferred onto and processed by the corresponding sub-components as required. Thus, inputs may be reacted to as soon as they are received.

However, in order to provide an output it may be necessary for the component to have obtained a predetermined memory location or the like, or be able to insert data into a bundle for transfer to the root component. In this instance, the root component typically generates a bundle and transfers this to the output of the component. This is then transferred onto the output of each of the sub-component which is to generate an output for transfer to the root server.

It will be appreciated that this mechanism allows the data to be generated and constructed

on the root server in desired memory locations.

In order for this to function correctly it is necessary for the chain to be passed from the output agents through any sub-components and back to the output agent. In order to achieve this, it may be necessary to include termination components as will be described in more detail below.

Hand-Off

As described above, hand-off of agents occurs to allow agents not explicitly involved in interactions, to pass on responsibilities to other sub-components.

An example of agent hand-off will now be described.

In order for agent hand-off to be performed correctly it is necessary for hand-off to be performed in accordance with a predetermined order. This is particularly important where bundles are involved.

An example of this will now be described with reference to Figures 43A-D. In particular Figure 43A shows a schematic including a component 1100 which includes three sub-components 1101, 1103, 1105 each which has respective agents 1102, 1104, 1106 adapted to be coupled to a bundle component 1107 via agents 1108, 1109, 1110. The bundle component 1107 includes an agent 1111 adapted to be coupled to an agent 1112 of the parent component 1100. Similarly, a component 1120 is provided which includes a similar schematic as shown.

In this Figure, the connections between the agents have not yet been implemented and are therefore shown as dotted lines.

Initially, as shown in Figure 43B, when the schematic is first built, with the respective component instances and corresponding agents being generated, the agents operate to connect as shown in Figure 43B. Thus, initially, the agents 1102, 1104, 1106 connect to the agents 1108, 1109, 1110, with the agents 1112, connecting to the agents 1132.

The agents 1112 and 1132 negotiate and determine they do not need to take any further part in the process, and in particular, they determine that they can hand-off to the agents 1111, 1131. In order to achieve this the agents 1111, 1131, will need to exchange the addresses of the agents 1111, 1131. As each agent can only connect to a single other agent, this is achieved by creating a temporary agent associated with each of the agents 1112, 1132. This is represented by the dotted lines in the agents 1112, 1132, in Figure 43B.

In this case, the agent and associated temporary agent are referred to an internal agent and an external agent. In the case of agent 1112, the internal agent will couple to the agent 1111, with the external agent connecting to the external agent of the 1132.

The internal agents may be generated by the component hosting the agent, with the external agent being generated by the parent component. Thus, in this example, the internal agent 1112 is generated by the component 1100, with the external agent 1112 being generated by an external component or the root server.

Each internal agent 1112, 1132 determines the address of the respective agent 1111, 1131. The addresses are then transferred between the external agents 1112, 1132, and transferred on to the agents 1111, and 1131 as required. Once the agents 1111, 1113 have obtained each others address, they can communicate directly, allowing these agents to negotiate directly with each other. The agents 1112, 1132 will then retire, as shown in Figure 43C.

In this instance, when agents 1111 and 1131 negotiate it is found that components 1107, 1127 can now perform their service. In particular, it will be determined that further hand-off can now be performed as the bundle and debundle components have corresponding inputs and outputs.

Accordingly, addresses of the agents 1108, 1009, 1110 will be included in a bundle, which is transferred from the agent 1111 to the agent 1131. This will be debundled with the addresses of the agents 1108, 1109, 1110 being transferred to the agents 1128, 1129, 1130 respectively. A similar process will be performed in the opposite direction. Once the agent addresses have been transferred, connections need to be formed between the agents 1108, 1109, 1110 and the agents 1128, 1129, 1130. In order to achieve this, temporary agent will need to be generated as described above, such that the external agents 1108, 1009, 1110 will be connected to the agents 1102, 1104, 1106, and the internal agents 1108, 1109, 1110 will be connected to the internal agents 1128, 1129 1130, as shown in Figure 43D. At this point the agents 1111 and 1131 can retire.

Once this is complete the agents 1108, 1128; 1109, 1129; and 1110, 1130 negotiate to allow the addresses of the agents 1102, 1122; 1104, 1124; 1106, 1126 to be exchanged, in a manner similar to that described above. Thus, for example, the address of the agent 1102 will be determined by the external agent 1108, transferred to the internal agent 1128, and then transferred via the internal agent 1128 to the agent 1122. Once this has been completed, hand-off can occur, with the agents 1108, 1109, 1110, 1128, 1129, 1130 retiring, and the agents 1102, 1104, 1106 communicating directly with the agents 1122, 1124, 1126 directly, as shown in Figure 43E.

It will be appreciated that the use of hand-off reduces interaction required by agents. Furthermore, when implemented in conjunction with bundling, allows hand-off of entire

bundles, corresponding to many agents. This thereby further reduces computational load and operator complexity due to the simplified schematic.

Connections

In all the above examples interaction between agents has been performed on a one-to-one basis. Thus, a single output from a component is connected to a single input on a subsequent component with each agent being adapted to interact with a single corresponding agent..

This vastly reduces the complexity of the system by ensuring that it is simple for agents to negotiate. In this case, components such as the DUP component described in more detail below, can be used to duplicate an output from an agent, allowing this output to be transferred to a number of subsequent agents, to thereby provide effective one-to-many connections. However, as an alternative, agent behaviour can be modified to provide one-to-many, many-to-one and many-to-many interactions to be performed between agents. Thus, for example, an output from a single component may be coupled to the inputs on several successive components. It will be appreciated that in this instance data provided at the output of a component may be provided to the inputs of several subsequent components simultaneously without the need for a connecting DUP component. In this instance, the agent associated with the output will need to negotiate with several agents simultaneously.

An example of this will now be described with reference to Figure 42.

Figure 42 shows a component 1030 having an agent 1031, which in this example is adapted to provide an output in decimal, binary or hexadecimal code, in that preferred order. The component 1030 is coupled to components 1032 and 1034 via respective agents 1033, 1035 adapted to receive data in decimal or hexadecimal and hexadecimal or

binary data forms as required. Accordingly, in this case it will be appreciated that the agents 1031, 1033, 1035 must negotiate to determine a common data format which in this case is hexadecimal code.

The component 1030 will then operate to provide an output in hexadecimal code format. Thus, the agent 1031 must provide the output in a least preferred format. In this case, if a third component 1036 is provided adapted to receive data via an agent 1037 which is only adapted to operate in decimal code it will be appreciated that no common data format can be found thus causing a build exception error to occur. Thus, it can be appreciated that providing from too many outputs, interactions will vastly complicate the negotiation and data transfer process.

Despite this, there are significant in providing for one-to-many, many-to-one and many-to-many connectivity between agents in that this will allow broadcast data to be provided from a single agent to a number of other agents, which can result in increases of efficiency in some circumstances.

Staged Construction

Staged construction can be used to allow dynamic components or dynamic schematics to be implemented. In particular, staged construction typically refers to when a designer deliberately lays out a schematic in a number of stages to control a difficult build then implements this when he is satisfied of progress of early stages. However, this can also be implemented at the component level.

In particular, in the case of dynamic components this allows the functionality of a component to be modified during implementation dependent on the results of earlier data manipulations. An example of this will now be described with reference to Figures 44A - 44C.

As shown in Figure 44A, a component 1050 having agents 1051, 1052, 1053, 1054 and 1055 is provided. The component is adapted to receive inputs via the agents 1051, 1052, 1053 and provide an output via the agent 1054. The component 1050 includes a sub-component 1060 having input agents 1061, 1062 and an output agent 1063. As shown the input agents 1061, 1062 are coupled to the input agents 1051, 1052, with the output agent 1063 being coupled to the internal agent 1055.

In use, the component 1050 is adapted to receive data via the agents 1051, 1052, 1053, manipulate these and then provide an output via the agent 1054. In this example, when data are received via the agents 1051, 1052 these are transferred to the component 1060 which manipulates the inputs and generates an output transferred via the agent 1063 to the internal agent 1055.

The internal agent 1055 is a particular type of agent known as a selector. In particular, the agent 1055 will operate to examine the payload received from the agent 1063 and then cause one or more schematics to be built depending on the data contained therein. Thus for example, if the payload received from the agent 1063 is a decimal number the selector agent 1055 may select a schematic 1064 containing a single component 1070 and cause this to be incorporated into the schematic of the component 1050 as shown in Figure 44B. Accordingly, agents 1071 and 1072 will be coupled to the agents 1053 and 1055 as shown, with an output agent 1073 being coupled to the agent 1054.

It will be appreciated that the schematic 1064 may have external agents, coupled to the agents 1071, 1072, 1073, which will need to hand-off as described above, to allow the agents 1071, 1072, 1073 to connect to the agents 1053, 1054, 1055, as shown. Similarly, as the selector agent 1055 cannot connect to two agents simultaneously, the connection will also require the creation of a temporary agent 1055, as shown by the dotted line in

Figure 44B.

In use, once the component 1070 has been incorporated into the specification the agent 1055 can operate to hand-off the agent 1063 to the agent 1072 in a manner similar to that described above, thereby allowing the component 1070 to manipulate data received via the agents 1071, 1072 as required. Manipulated data can be provided via the agent 1073 as will be appreciated by a person skilled in the art.

However, the agent 1055 may determine that the output provided by the agent 1063 is in a hex format in which case the selector agent 1055 must select a different schematic1065. An example of this is shown is shown in Figure 44C, in which case the incorporated schematic includes the component 10-70 coupled to an additional component 1080 for converting the hex number into a decimal format. Thus the schematic shown in Figure 44C includes the component 1080 having an input agent 1081 coupled to the internal agent 1055 and an output agent 1082 coupled to the input 1072 of the component 1070.

In this instance, the internal agent 1055 will cause the schematic to be built before transferring the payload to the agent 1081, allowing the agent 1080 to convert the data into a decimal format before it is transferred via the agent 1082 to the agent 1072. Again, this may require the formation of a temporary agent 1055, as shown. The remainder of the operation will be as described above.

It will therefore be appreciated that the schematic which is used to control the manner operation of the component by the agent 1055 once some initial processing has already been completed. In this case it is not therefore possible for the entire build process to be completed simultaneously and instead this therefore requires a staged construction.

In the above process, the selector agent therefore needs to analyse the output provided by

the component 1060 in order to determine the subsequent schematic 1064, 1065. Thus during the build process, the selector agent 1055 will operate to negotiate with the agent 1063 in normal way. When this has been completed and the component 1060 performs the service embodied therein, the agent 1063 will supply the output to the selector agent 1055 as a payload. The selector agent determines the output contained in the payload, and then selects the subsequent schematic 1064, 1065 as required. The subsequent schematic can then be built and implemented in the normal way in which any sub-component would be implemented.

It will be appreciated from this that the agent 1055 may terminate the connection with the agent 1063 before the schematic is selected in which case a temporary agent 1055 will not be required.

In order to implement this form of component the entity implementing the component 1050 will therefore need to modify its mode of operation. In particular, if the schematic of the component 1050 is predefined, as in normal circumstances, when the entity receives a purchase order it will generate a component instance 1050. This component instance will in turn determine that additional component instances are required for any sub-components such as the components 1060 and 1070. Accordingly, respective component instances are initiated for the components 1060 and 1070 as defined in the schematic. It will be appreciated that these component instances may be implemented on different component servers to the component 1050, and indeed may be implemented by different entities.

In any event, at this point, the agents 1063, 1071 will negotiate directly prior to any data being processed. Thus the component instances 1060 and 1070 are ready to receive data prior to any data actually being processed.

In contrast to this in the current situation when the entity receives a purchase order for the component 1050 it will initially initiate a single component instance, corresponding to the component 1060. In this situation the component instance 1060 will operate to perform data manipulations in the usual way providing the output to the schematic selector agent 1055. The selector agent 1055 will then operate to determine a schematic needed to implement the remaining service, or data manipulations as required.

Thus in the case of the schematic shown in Figure 44B the schematic selector agent 1055 will select a schematic resulting in the use of a component instance corresponding to the component 1070, whereas in the case of the schematic of Figure 44C the selector agent 1055 will initiate the schematic 1065 resulting in the use of two respective component instances corresponding to the components 1080 and 1070. .

Staged construction may be implemented either automatically or manually depending on the preferred implementation.

In the case of manual operation, the internal agent 1055 may be adapted to provide an output to the entity operating the component 1050, such as an indication of the data received in the payload from the agent 1063. An operator will examine the output and construct the remainder of the component schematic shown for example in Figures 44B, 44C as required. The user defines the schematic and then operates to build it, causing respective component instances to be generated in the normal way. Thus, the agent 1055 will perform hand-off as described above. Once the components have been implemented and agent negotiation completed, the component 1050 then allows the remaining data manipulations to be completed as required.

Alternatively, in the case of the automatic operation the agent 1055 will trigger the selection of the remaining components and their interconnections automatically. It will

be appreciated that this generally has to be achieved in accordance with predefined schematics which are selected from a predetermined list stored in the memory, or the like. This selection can therefore use the data received from the agent 1063 to access a Look-Up Table (LUT) and determine the required schematic.

In order for this to function correctly, it will need to be ensured that schematics are available to handle any data that may be potentially provided at the agent 1063, or the overall build process may fail, or have to divert to a manual operation.

Thus, in the above example, the build process is completed in two stages, although any number of stages may be provided. In any event, the technique of staged construction. allows a wide range of functionality to be achieved.

In particular, the use of stage construction can apply to overall schematics. Thus, users may define a schematic including a staged construction. In this case, the user can define three or four different schematics with the final schematic used depending on results obtained part way through the manipulation. In the case in which this is performed, any components that potentially will perform data manipulation after a selector agent will not be constructed until the selector agent has made a decision on the schematic to be used.

This decision may be made in conjunction with input from the user. Thus, the user can partially define a schematic to produce a stage output. Once the stage output has been obtained, the user can complete the remainder of the schematic in response to the particular output obtained.

It will be appreciated that in this instance, purchase orders are not issued for the components involved in the second stage of construction, until the first stage has been completed and the desired output obtained.

This allows users to perform data manipulations and build executable code in stages. This in turn reduces the expense incurred if the build process fails, as well as providing users opportunity to optimise schematics since the number of variables to handle may be reduced.

Staged construction also allows dynamic components to be implemented. In this case, the entity may define a component that requires staged construction in order to complete, with this fact being transparent to the user. The component will interact with other components in the schematic during the build process, by offering the external agents, such as the agents 1051, 1052, 1053, 1054, in the normal way. This allows other components to be constructed as normal, with the output from the component 1050 only being provided once the second internal stage of construction is complete.

As a result, it is generally the entity that issues any purchase orders required to implement the components required in subsequent stages of construction, and it will therefore be appreciated that this may be performed automatically in some cases. The entity server may therefore act as a root server for the second stage of construction, particularly if the components used in the second stage of construction are provided by another entity.

Prototype Components

In the examples above the components are predefined, even in the case of dynamic components. This is because the nature of the input and output data is predetermined. This means that when the user selects a component from the forum they are aware of the specific input and output requirements of the component.

Thus, in the example described above with respect to Figures 44A -44C the component may be dynamic and this may allow for example the component to receive inputs having

different formats. However, it is still necessary for these formats to be predefined thereby providing the user with only limited options for implementation.

As an alternative, an entity may provide a prototype component which is situated on the forum. The prototype component will not include a defined component schematic and similarly will not include finalised input and output specifications. Instead, the prototype component will include an indication of the functionality that may be provided. Accordingly, the user may select a prototype component which is capable of combining inputs.

At this stage the nature of the inputs and/or outputs, and in particular the data formats that can be handled are undefined. Accordingly, when the user selects the prototype component and includes this in a schematic it will be necessary for the user to provide the entity with an indication of the data formats that they wish to combine. This procedure can be performed in a number of ways.

In a first example when the user selects the component and attempts to add this into a schematic the user will be prompted to provide input and/or output specifications they desire for the component, with this being transferred to the entity for review. The entity then assesses if it will be possible to provide the respective service in accordance with the requirements. If so, and the component can be implemented, confirmation of this is sent to the user together with defined input and output specifications for the components agents as required. The specifications will then be used to finalise the component allowing it to be added into the schematic.

This form of prototype component will hereinafter be referred to as a conditional prototype as this requires confirmation from the entity that the component can be implemented before it may be incorporated into the schematics.

As an alternative however if the entity is confident that they can perform the data manipulation service embodied in the component regardless of the input and output specifications required, then the prototype component can be provided as unconditional prototype.

In this case, the prototype component may simply be incorporated into the schematic immediately. In this instance, the agents of the component will be adapted to communicate with agents of other components and determine input and output specifications therefrom. The agents will therefore effectively accept any input or output options selected by corresponding agents during the build process when the agents negotiate in the normal way.

Agent negotiation is basic in that the agent of an input of one of the unconditional prototype components will simply indicate that it can receive data from an output in accordance with the preferred format specified by the output agent. Similarly, the output will simply indicate that it can provide data in the format required by the agent associated with the input of a subsequent component.

Accordingly, the input and output specifications of the unconditional prototype are therefore effectively determined in accordance with the input and outputs specifications required by other adjacent components in the schematic.

It will therefore be appreciated that if two conditional prototypes components are interconnected, common specifications may not be defined automatically. In this instance, some user input either by the user creating the schematic or by the entities implementing the components will be required in order for the schematic build to be successful.

In any event, in the case of unconditional prototype components, when the component is to be implemented the entity will simply operate to receive the data and perform the required data manipulations regardless of the format. It will be appreciated that this may be performed automatically if a suitably flexible component can be defined.

Fundamental Components

As described above, the system allows components to be combined hierarchically to allow a complex series of data manipulations to be performed. Thus complex components can be constructed using basic fundamental components, which may be provided on the forum.

The fundamental components represent basic data manipulations that will be required in most schematics, such as logic manipulations, or the like, and are typically implemented automatically utilising suitable executable code.

Examples of some of fundamental components will now be described. In the following examples the executable code actually used to implement the components will not be described as this is generally straight forward and well within the scope of any programmer as will be appreciated by a person skilled in the art. In any event, it will also be appreciated that different forms of fundamental component implementation may be provided.

The fundamental components now being described will be broken into a number of different groups depending on the functionality.

Agent Components

The following components are adapted to provide functionality for agents.

*Bundle*

This component operates to bundle agent connections together into a single agent connection. The bundle component is used primarily for two reasons.

Firstly, the use of the agent bundle vastly simplifies the process of creating schematics. In particular, it would be typical for a component implemented by an entity to need to react through a large number of inputs and outputs with other components. However generally the number of inputs and outputs will have to interact with corresponding inputs and outputs on other components. Accordingly, the agents can therefore be combined together into agent bundles so that only a single connection between the two components needs to be defined in the schematic, as described above.

In addition to this, the use of agent bundles vastly simplifies the build process. In particular, it will be appreciated that hand-off of agents often occurs, and by handing-off bundles as opposed to many individual agents, this vastly reduces the interaction required.

A further benefit is that bundling allows hierarchical bundling to be performed as described above.

A range of different bundling components may be provided including different numbers of inputs and outputs and an example of four of these are shown in Figures 45A-45D. Descriptions of the agents are as set out in table 5 below:

Table 5

| Agent | Name | Description | Agent Type |
|---|---|---|---|
| NI | name in | name of ingoing bundle | Name |
| NO | name out | name of outgoing bundle | Name |
| 1, 2, 3, 4, 5 | inputs | agents to be bundled | N.A. |
| E | exception | boolean | non-typed |
| B | bundle | output | Bundle |

*Agent Split*

This component is used to allow incoming and outgoing messages to an agent to be split.

An example is shown in Figure 45E. In particular, the component includes a single combined input/output agent I/O, together with an input agent I and an output agent O.

Normally an agent is a single point of connection such that it can receive incoming data and supply outgoing data. However, sometimes this is not ideal and it is preferable that separate input and output agents are provided. Accordingly, the agent split connects to transfer outgoing data through the agent O and the agent I/O, with incoming traffic being transferred via the agents I, I/O. Functionality is generally reversible. Descriptions of the agents are as set out in table 6 below:

Table 6

| Agent | Name | Description | Agent Type |
|---|---|---|---|
| I | Input | Connected to input | Non-typed |
| O | Output | Connected to output | Non-typed |
| I/O | Input/output | Connected to input/output | Non-typed |

*Check Exception*

This component operates to check an input string for the word exception. The component shown in Figure 45F includes an input agent I, an output agent O and an exception agent E.

In this case the input agent I receives an input string, which the component would assess to determine if it includes an exception word. If an exception is not found then the input is transferred to the output agent O and the exception agent operates to provide a false indication at the output E. If the input string is an exception or includes an exception then the input is blocked from the output and the exception agent operates to provide a true output at the output E. Descriptions of the agents are as set out in table 7 below:

Table 7

| Agent | Name | Description | Agent Type |
|---|---|---|---|
| I | Input | input string. | Non-typed |
| O | Output | connected to input. | Non-typed |
| E | Exception | Exception detected, boolean. | Non-typed |

*Diode Component*

The diode component includes an input agent I coupled to an output agent O in a one way direction as shown in Figure 45G. This is used to ensure that the communication between agents only occurs in one direction.

It will be appreciated that the diode component may be provided with a number of outputs as required.

*DUP*

The dup components includes an input agent I coupled to two or more output agents O, as

shown for example in Figures 45H, I, J.

In use, an input string received at the input agent I is duplicated and supplied by each output agent O as required. This allows data from a single component to be broadcast to a multiple number of components as required. It will be appreciated that this provides a way of effectively allowing one to many connectivity between agents whilst still requiring each agent to interact with one other agent only.

An interesting property of DUP is that should one of the outputs receive a terminate from its peer then the DUP simply hands off the input I to the remaining output O and DUP retires. An example of this is shown in Figures 45L and 45M. In this example, a DUP component couples an agent A to the agents B, C. If the agent C retires and terminates the connection as shown in Figure 45L, then the input I and remaining output O can perform hand-off, allowing the agents A B to connect directly as shown in Figure 45M.

*Wait Component*

As shown in Figure 45K, the wait component includes an input agent I, an output agent O and a trigger TRIG. The component is adapted to receive one input and then transfer this to an output upon receipt of a trigger.

Again the functionality is as shown in table 8 below.

Table 8

| Agent | Name | Description | Termination |
|---|---|---|---|
| I | Input | pass through agent | follow |
| O | Output | pass through agent | follow |
| Trig | Trigger | trigger on reception of 'true' payload. | follow |

Conditional Components

Conditional components are provided to implement basic mathematical functionality. As shown in Figures 46A to 46E, each component includes two inputs I1, I2, an exception output E, and an output O. In this case, the components are adapted to compare strings received at the inputs I1, I2 and provide an output at the output O, or an exception at the exception output E, in accordance with the results of the comparison.

In general the components will perform mathematical functionalities, as set out below in Table 9.

Table 9

| Component | Description |
|---|---|
| EQ | Determines if input strings are equal |
| LT | Determines if input string I1 is less than input string I2 |
| LTE | Determines if input string I1 is less or equal to input string I2 |
| GT | Determines if input string I1 is greater than input string I2 |
| GTE | Determines if input string I1 is greater than or equal to input string I2 |

Construction Management

In order to aid with construction management, a number of different components are provided as set out below.

*Build*

A number of build components are provided as shown in Figures 47A to 47D. In particular, the build components are adapted to add code into a build chain as described below. The functionality of the build components is set out in the table 10.

Table 10

| Component | Description | Agent | Description |
|---|---|---|---|
| Build 1 | Adds code to the build chain and allows memory usage to be specified before entering the code | Code | Code in hex |
| | | Malloc | Number of bytes to be supplied |
| | | Ac'bility | Unused |
| | | Build | Build bundle |
| Build 2 | Adds code to the build chain and allows a memory address to be obtained from a previous component | Code | Code in hex |
| | | Address in | Address input |
| | | Address out | Address output |
| | | Ac'bility | Accountability |
| | | Build | Build bundle |
| Build 3 | Adds code to the build chain and inserts memory addresses internally | Code | Code in hex |
| | | Build | Build bundle |
| Build 4 | Adds code to the build chain and allows a memory address to be accessed from a previous component | Code | Code in hex |
| | | Build | Build bundle |
| | | Addr out | Address out |

The internal schematics of the build components are shown in Figures 47E to 47H. As shown, the build components each contain a BUN3 bundle component, which is adapted to construct a build bundle, as described below. In addition to this, each of the schematics include additional components as shown, with these components having functionality described above or below.

*ELF INIT*

The ELF INIT component accepts code from a build component in hexadecimal format and produces an executable binary program. To create an executable program it is necessary to save the payload output from the ELF agent and then make it executable. An example of an ELF INIT component is shown in Figure 47I, with an internal schematic being shown in Figure 47J. Again, as operation of the ELF INIT component in accordance with the schematic set out in Figure 47J is self-explanatory, this will not be described in any further detail.

In this case, the agent CODE accepts the code from the build component, with the executable program being output from the agent ELF. Operation of this will be described in more detail below.

*Cat*

The cat component shown in Figure 47K operates to concatenate build chains together. Accordingly, the cat component includes two input agents b1, b2 for receiving respective build chains and a single output agent BUILD for providing a concatenated chain.

The internal structure of the concatenate component is shown in Figure 47L. As shown, the cat component uses two bundle components bundle1, bundle2 to split the incoming chains into three main strings, namely an address in, an address out and code which are provided respectively at the agents IN, OUT, CODE, as shown. A string cat component, described in more detail below, is used to receive the code from the components bundle1, bundle2 and concatenate this into a single string. This string is then output to a third bundle component bundle3 via an output agent O.

In addition to this, the agent IN of the component bundle1 is connected to the agent OUT of the component bundle2 with the agent IN of the component bundle2 being connected

to the agent OUT of the component bundle3 and the agent OUT of the component bundle1 being connected to the agent IN of the component bundle3. This ensures that the chains are passed through the schematic unbroken as described above.

In addition to this, the component bundle3 is connected to an ELF INIT component. A build bundle is supplied to the component bundle3 by the component ELF INIT, and is unbundled, with the address being transferred via the agent IN of the component bundle3 to the agent OUT of the component bundle1. It then leaves the cat in a build bundle via an input IN1 and is transferred to an external component for modification as required.

Once modified the build bundle is transferred back to the component bundle1, and is unbundled. A new or modified address included in the build bundle is transferred via the agent IN of the component bundle1 to the agent OUT of the component bundle2. Again, the address is bundled in a build bundle and transferred via an agent IN2 to an external component. The external component returns a further modified build bundle including code to be concatenated with the code received via the agent IN1.

At this time the build bundle enters back through the agent IN2 and is unpacked in component bundle 2, with the modified address being passed to the agent OUT of the component build3.

The code from the build bundles is transferred via the agents CODE of the components bundle1, bundle2, to the cat component.. The cat component concatenates the code, transferring the concatenated code to the component bundle3. The component bundle3 creates a build bundle including the concatenated code and the modified address outputting this to the component ELF INIT, where the new address with all the changes is registered.

Accordingly, the cat component is used to connect components with a build output. This is includes the build components outlined above.

A further example of the use of this is shown in Figure 47M. In this example, three build components are interconnected via two cat components to an ELF INIT component. The flow of data through the schematic is shown and will not therefore be described in detail.

In order for a component to construct executable code during the build process it is necessary for the component to be able to obtain access to the construction site. Accessing the construction site is typically a very complex process requiring the manipulation of many agents. The purpose of the construction site is the erection of the binary executable and often most, if not all of the components involved in the construction must interact with the construction site in some way.

An example of the most simple form of interaction will now be described with reference to the example shown in Figures 48A.

As shown in Figure 48A, three components 1151, 1154, 1157, each having a respective input agent 1152, 1155, 1158 and respective output agent 1153, 1156, 1159, are connected as shown to agents 1160, 1161. The components are provided to produce executable code, and operate by inserting code fragments into a build bundle.

In particular, a header file is provided by the agent 1160 to the agent 1151. This typically provided an agent payload. When the component 1150 receives the header file it appends its code to the end of the header file and forwards it on via the agents 1153, 1155 to the component 1154. Again, this will typically be in the form of an agent payload.

In any event, component 1154 receives the executable code fragment appended to the

header file, and appends its own executable code fragment. Alternatively, or additionally the component 1154 may modify the code fragment provided by the component 1150. In any event, the header file including the appended code fragments is transferred on to the component 1157 in a similar manner for further code fragments to be added. The process proceeds until all required code fragments are added to the header file and the resulting executable code is supplied to the agent 1161.

An example of the structure of the header file and appended code fragments is shown schematically in Figure 48B, with the reference numeral being indicative of the component generating the respective code portions.

It will be appreciated that many more details are required to effectively construct an executable program. RAM allocation, processor register allocation, physical address details and general global details of the target construction site.

In order to supply all the above data it is necessary to expand the simple single agent connections as described in the first example to include additional agents, and indeed additional components. Considering the example shown in Figure 48C, where four sets of agents form several unbroken chains allowing each component in turn to add code, allocate memory, reserve or relinquish a CPU register etc. In this case, the file header is received from and returned to a single component 1063. It will be appreciated that interconnecting all the agents for al the components would be time consuming and would make the schematic unwieldy.

In the example, each of the components 1151, 1554, 1157 has a number of inputs agents and a number of output agents. These can be replaced by a single agent so that the respective component can access the chain involved in code construction or memory allocation using a single agent.

All these related agents then can be bundled into a single build bundle. Through the use of the components outlined above, the allows the functionality of separate agents to b e retained. Thus the schematic shown in Figure 48C, can be replaced with the schematic shown in Figure 48C. In this case, components 1164, 1165 are CAT components with the component 1163 being an ELF INIT component.

Further components can be added to simplify the processing internal to components 1151, 1154, 1157. These additional components effectively shield the components 1151, 1154, 1157 and all other components from the details of the build bundle. An example of this is provided by the build bundle described above, which allows a bundle received by the component to be debundled. An example of this is shown in Figure 48E, with the component 1166 being a build component, which received the build bundle at the agent 1167, presenting respective bundles or agents at the agents 1168, 1169, 1170.

It will be appreciated that if each of the components 1151, 1154, 1157 includes a respective build component, the flow of the file header becomes similar to that describes above with respect to Figure 47M.

Thus the component ELF INIT can start the process by supplying a header into the top of the chain and allow each component connected to the chain to append (using some suitable component) their code until the executable is constructed. Alternatively the ELF INIT component can wait for the outcome of the chain and prepend a header forming the ELF executable format as required by the operative system.

The result is a tree of components of sub-components connected by an unbroken line facilitating construction.

Arbitrary length chains can be easily constructed while hiding the complexity. In any event, this allows the supplier of the build component to expand the Build Bundle into hundreds or thousands of agent's as required to coordinate even complex construction sites.

In addition to the functionality described above, a component may export part of the build bundle to another component to insert code into the relevant section.

An example of this is shown in Figure 48F. In particular, this system includes three components 1200, 1201, 1202, adapted to generate respective executable code fragments. In this example, the components 1200, 1202, include respective sub-components 1203, 1204, 1205, 1206, 1207, also adapted to generate code fragments. These components, 1200, ... 1207 are connected via a number of build cat components 1208, 1209, 1210, 1211 to an ELF INIT component 1212. In use the header file is passed in turn to the component 1200, and hence to the components 1203, 1204, 1202, before being passed to the component 1201 and hence the components 1205, 1206, 1207.

Accordingly, this results in the header file being appended with code fragments as shown in Figure 48G.

In this example, if the link between the component 1200 and the component 1202 is created dynamically then the tree structure can be converted to a mesh structure, by creating links at build time.

<u>Conversion Components</u>

A number of conversion components are provided for converting data formats. Examples of two of these are shown in Figures 49A and 49B.

In particular, Figure 49A shows a bin2hex component adapted to receive a binary string at the input agent I and output a hexadecimal string at the output agent O. Similarly Figure 49B shows a h2b component adapted to receive a hexadecimal code at the input agent I and output binary code at the output agent O. In this example, the h2b component also includes an exception agent for providing an indication of any exception errors during processing.

System Components

A number of system components are provided for controlling system input and output.

File Components

File components are used to allow data from a file to be loaded into an agent payload. A load component is shown in Figure 50A, with a save component being shown in Figure 50B. The agents and their functionality for the components are as set out in Table 11.

Table 11

| Component | Agent | Description | Agent Type |
|---|---|---|---|
| LOAD | start | trigger load operation on receipt of 'start' payload | non-typed |
| | File | File name | non-typed |
| | Exception | Exception | non-typed |
| | Data | data read from the file loaded | non-typed |
| | Finish | indicate load is complete, with 'true' | non-typed |
| SAVE | Start | start save with any payload | non-typed |
| | File | File name | non-typed |
| | Exception | Exception | non-typed |
| | Data | Data to save, byte array | non-typed |
| | Finish | indicate save is complete, with 'true' | non-typed |

The load component operates to load data by reading a file specified to the agent FILE and provides an output via the agent DATA. The agent LOAD reads the file from the local hard-drive of the machine running the load component and then inserts this into the payload as required.

The save component saves any payloads received in the agent DATA to the file specified by the agent FILE. The save component saves over the file if it already exists. The save component saves the file to the local hard-drive of the server running the save component.

Logic Components

A number of logic components are provided to perform logical manipulations as will be appreciated by a person skilled in the art. Basic logic manipulations include AND, NAND, NOT, OR, NOR, XOR and, XNOR functionalities.

Examples of the components are shown in Figures 51A to 51G. Again in this example each logic component includes two agent inputs I1, I2 (only a single input agent I in the case of the NOT component) an agent output O and an exception agent E. This allows input strings received at the agents I1, I2 to be logically combined to produce an output at the agent O as will be appreciated by a person skilled in the art.

The logic components can be combined to produce more complex logic components as shown for example by the AND3 and AND4 components in Figure 51H, 51I. In particular the AND3 and AND4 components are formed from a number of respective logic components as shown in the internal schematics shown in Figures 51K, 51L. It will be appreciated that the logical components can therefore be combined in the similar way as normal logic gates.

<u>Protocol Components</u>

Protocol components are provided to check whether payloads have a particular protocol.

*ISProtocol Component*

An isprotocol is shown in Figure 52A. This component checks to see whether a payload received via an input agent IN is an XML protocol agent. This is achieved by having the component examine the input string and determine if it includes "P" tags required to define the respective protocol.

An indication that the correct protocol is output via an agent OUT. It will be appreciated that the internal schematic of this will vary depending on the respective protocol being detected, although an example of the general structure being shown in Figure 52B, with the system using a FIND TAG component to locate the tag within the received XML payload.

<u>String Components</u>

A number of string components are provided for performing actions on strings received at input agents.

*Cat component*

The cat component operates to concatenate strings received at agents I1, I2 and provide a concatenated output via the agent O as shown in Figure 53A.

*Find Tag component*

The find tag component operates to find a field having a specified name tag in an XML payload.

As shown in Figure 53B the find tag component includes an agent XML for receiving an XML string, an agent TAG for finding a tag to be located, an exception agent E, for providing an indication of an exception, an agent REM, and an agent OUT for providing the tagged data.

Thus, the find tag component performs a linear search through the XML payload for the tag specified by the tag agent. The data found between the start and end tag is sent to the "out" agent. The remainder of the document is sent to the remainder agent.

If the string for the tag is unsuccessful or there is a parsing error then an exception is generated resulting in a true output on the exception agent E. In this case no output is sent from the agents OUT, REM.

*STRCMP Component*

A STRCMP (string comparison) component is shown in Figure 53C.

The STRCMP component compares input strings received via agents I1, I2 and provides an output via the agent O. In particular, the STRCMP component will output a true indication via the agent O if the strings match.

*Unique Component*

The unique component creates a unique string each time it is called generally this will occur in sequential manner and is used to allow identifiers or the like to be assigned.

An example is shown in Figure 53D, and includes an output O for supplying a unique twelve digit number.

Persons skilled in the art will appreciate that numerous variations and modifications will

become apparent. All such variations and modifications that become apparent to persons skilled in the art, should be considered to fall within the spirit and scope that the invention broadly appearing before described.

Dated this twenty second day of April, 2003

**CODE VALLEY PTY LIMITED**

By their Patent Attorneys

**DAVIES COLLISON CAVE**

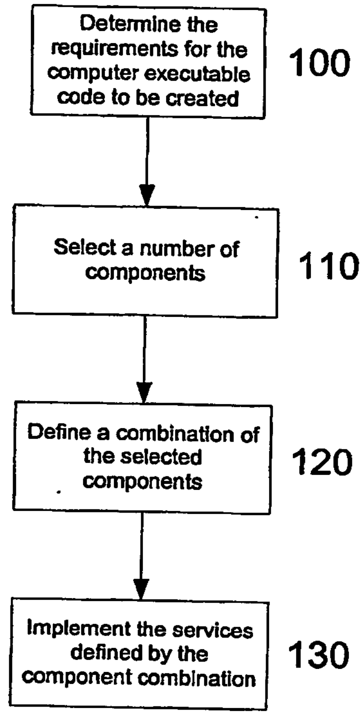

Determine the requirements for the computer executable code to be created
100
Select a number of components
110
Define a combination of the selected components
120
Implement the services defined by the component combination
130

Fig. 1

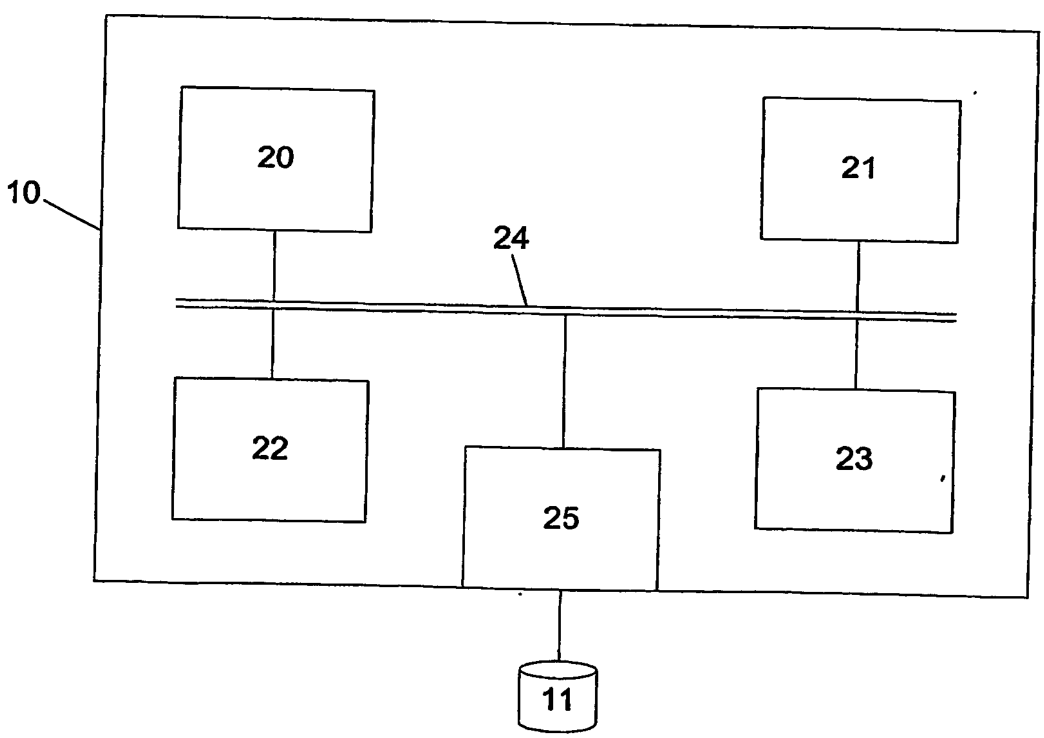

10
20
21
24
22
25
23
11

Fig. 2

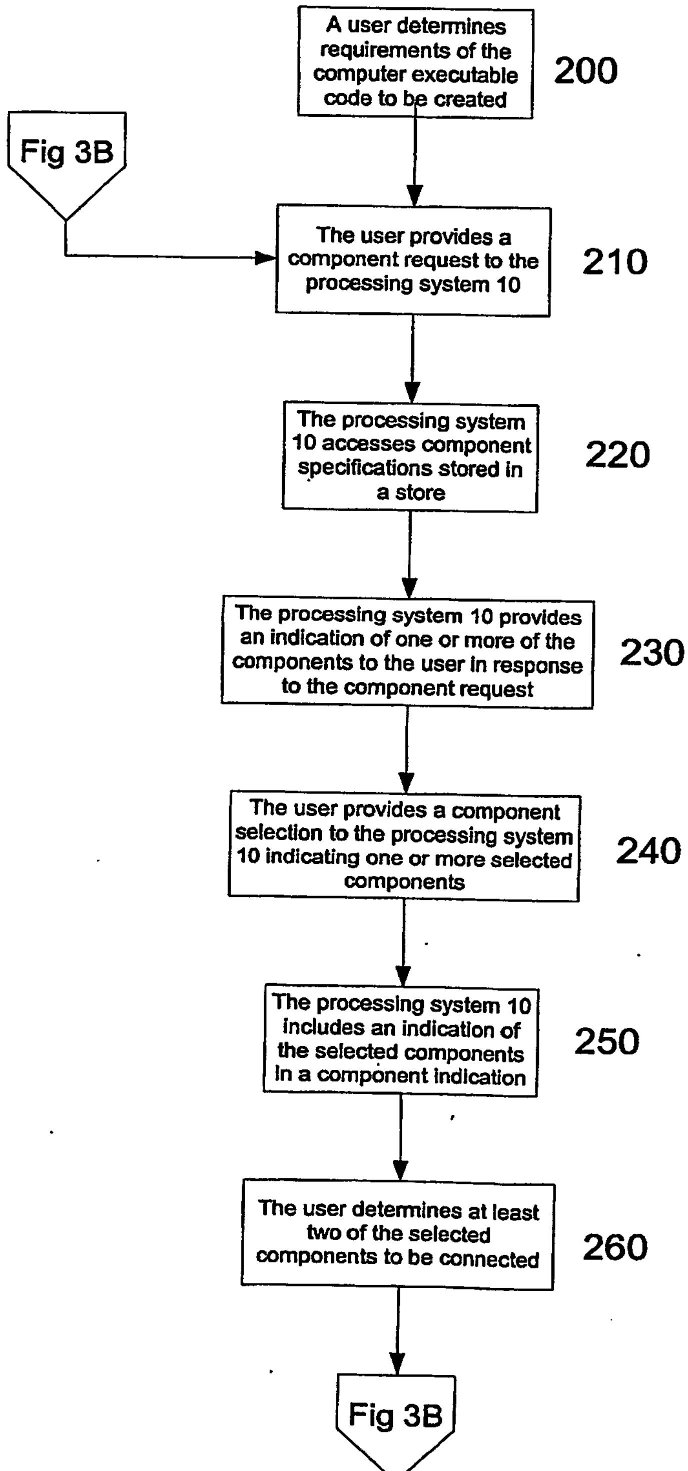

A user determines requirements of the computer executable code to be created
200
Fig 3B
The user provides a component request to the processing system 10
210
The processing system 10 accesses component specifications stored in a store
220
The processing system 10 provides an indication of one or more of the components to the user in response to the component request
230
The user provides a component selection to the processing system 10 indicating one or more selected components
240
The processing system 10 includes an indication of the selected components in a component indication
250
The user determines at least two of the selected components to be connected
260
Fig 3B

**Fig. 3A**

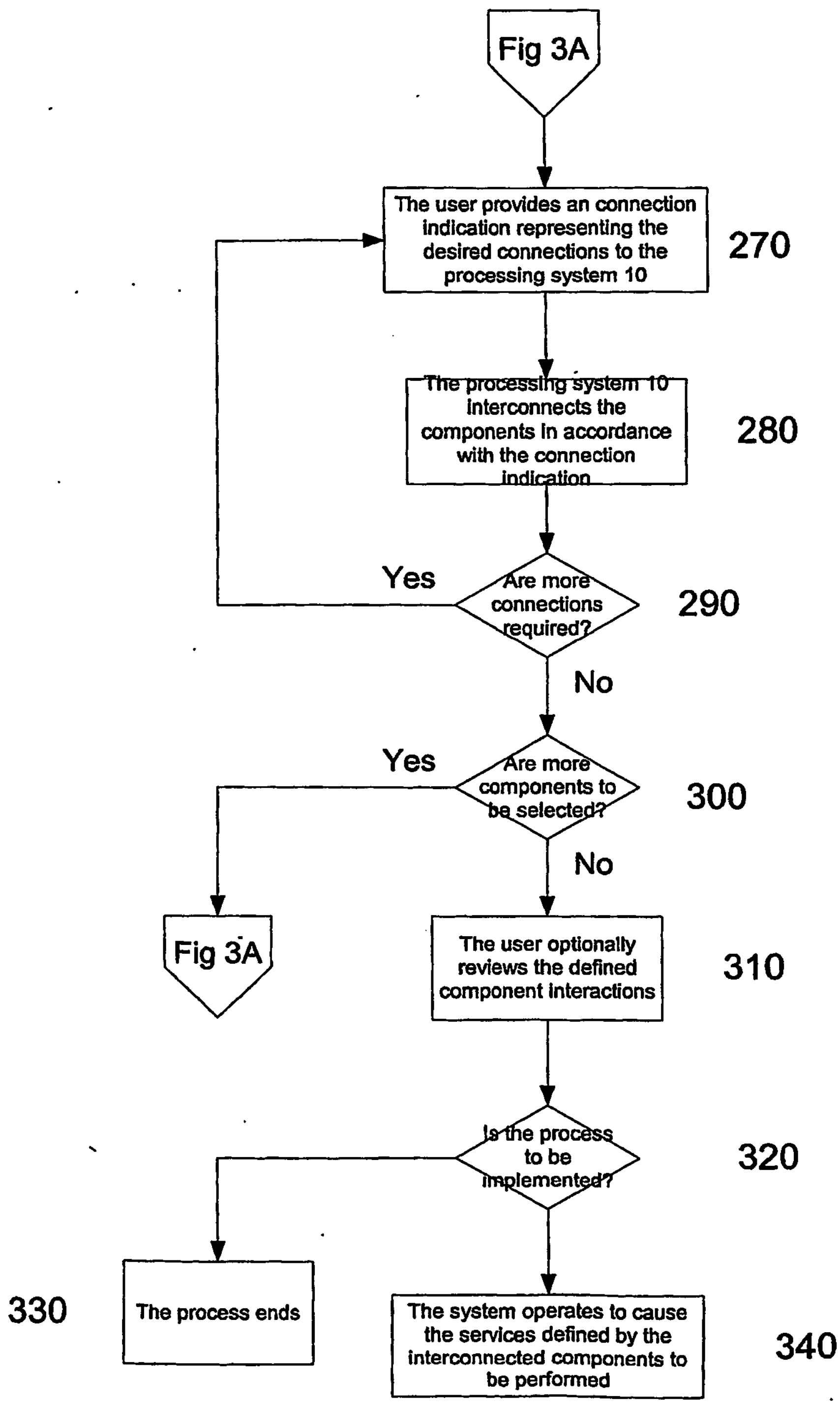


Fig 3A
The user provides an connection indication representing the desired connections to the processing system 10
270
The processing system 10 interconnects the components in accordance with the connection indication
280
Yes
Are more connections required?
290
No
Yes
Are more components to be selected?
300
No
Fig 3A
The user optionally reviews the defined component interactions
310
Is the process to be implemented?
320
330
The process ends
The system operates to cause the services defined by the interconnected components to be performed
340

Fig. 3B

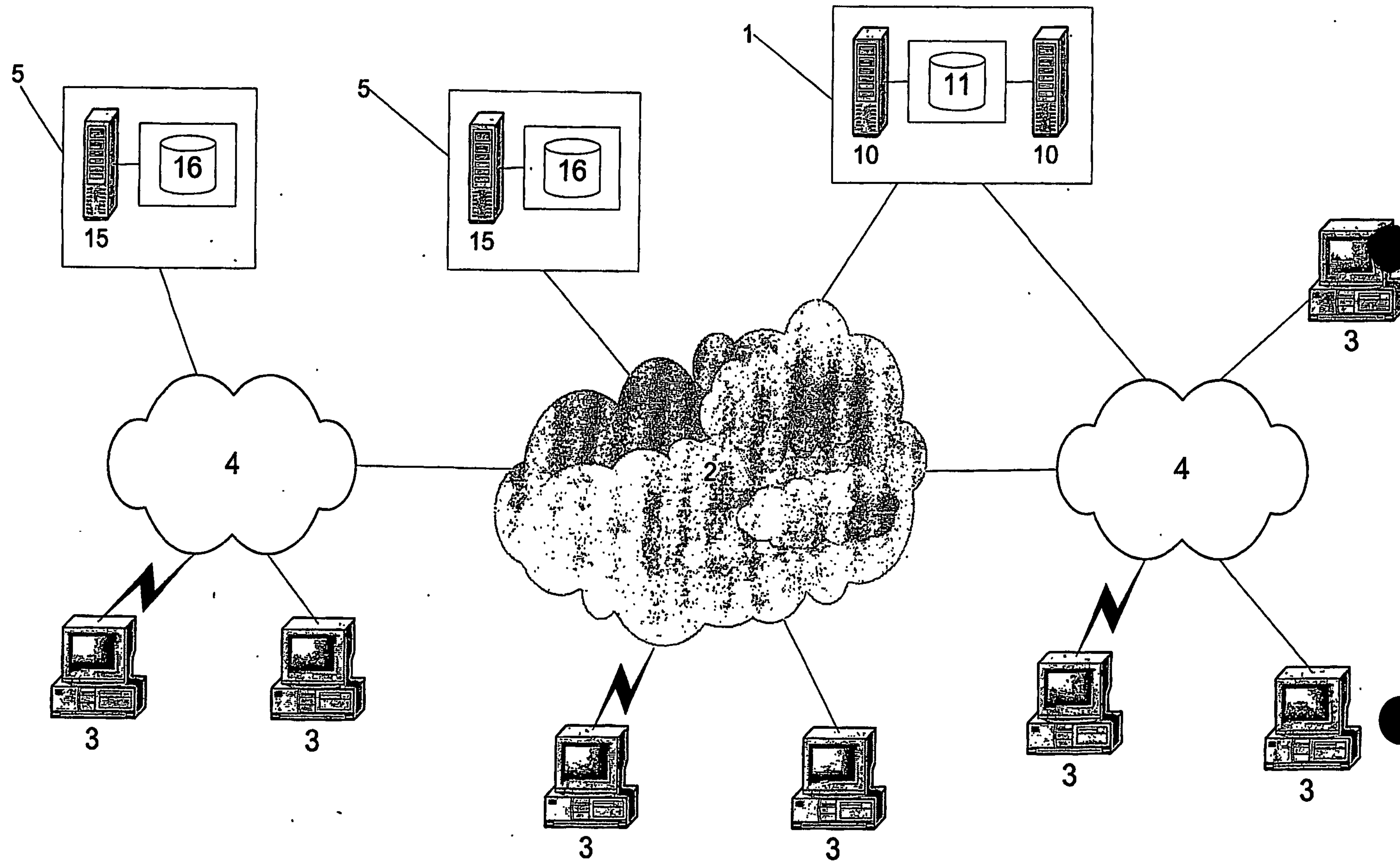



Fig. 4

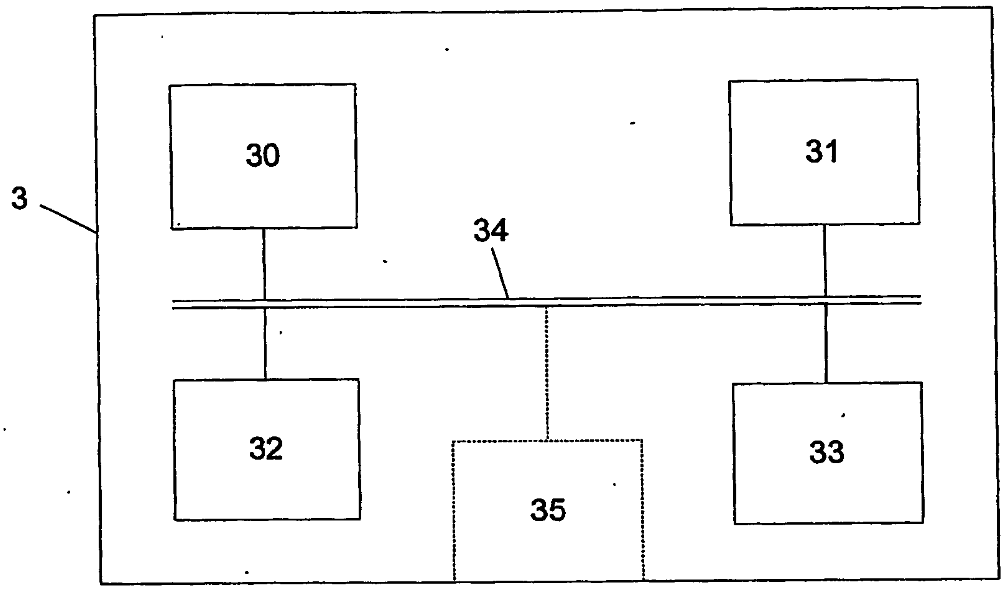



Fig. 5

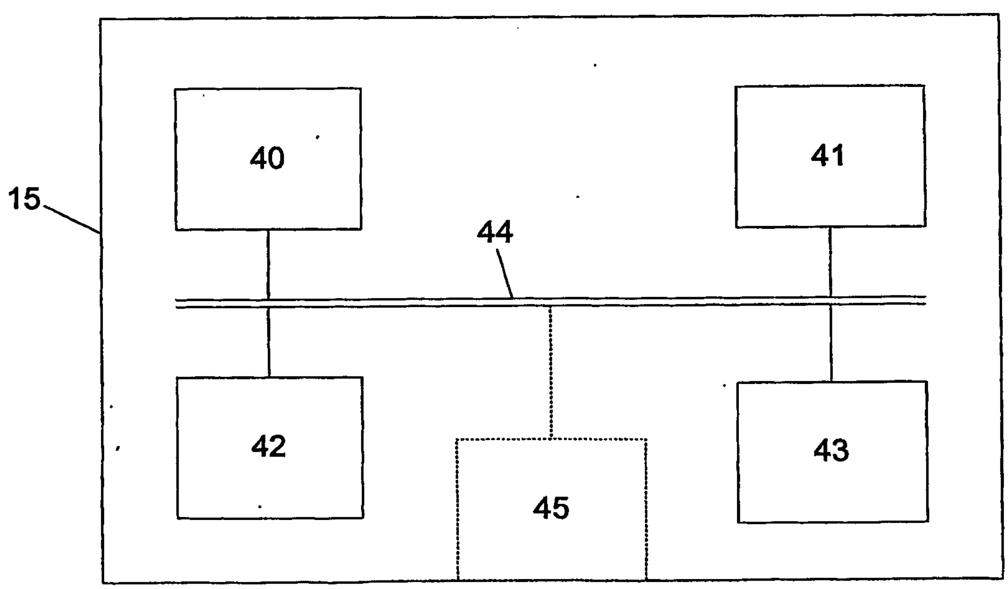



Fig. 6

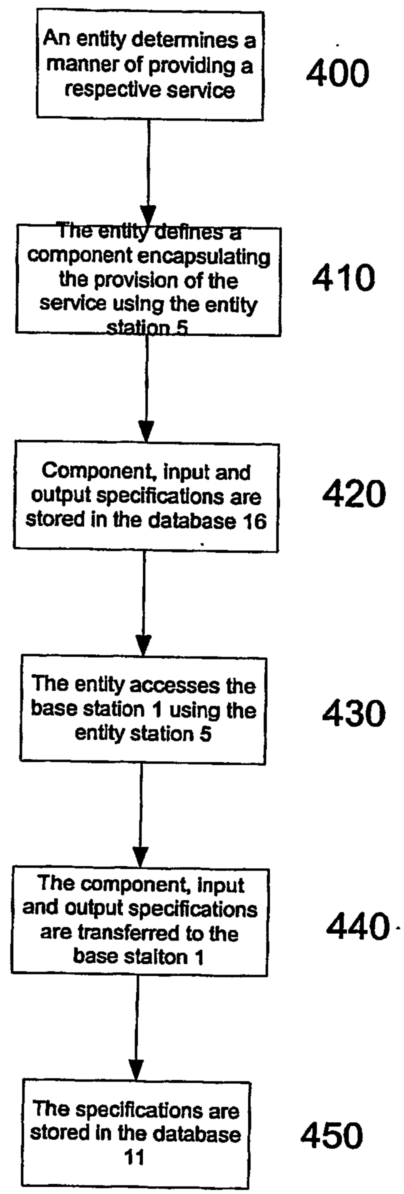
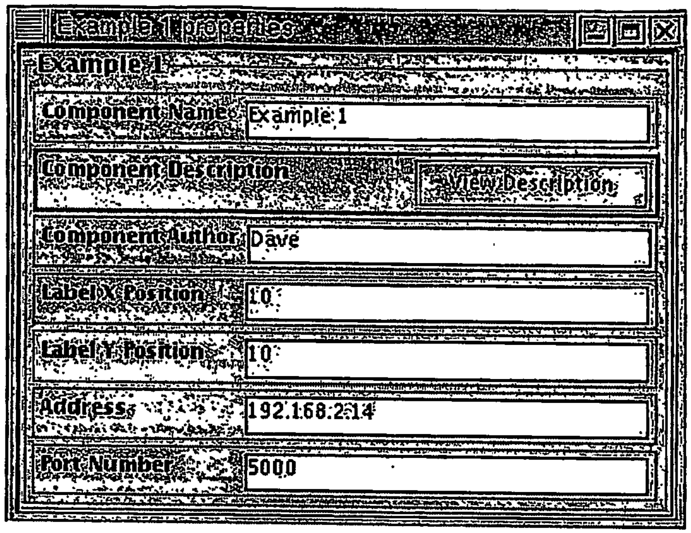
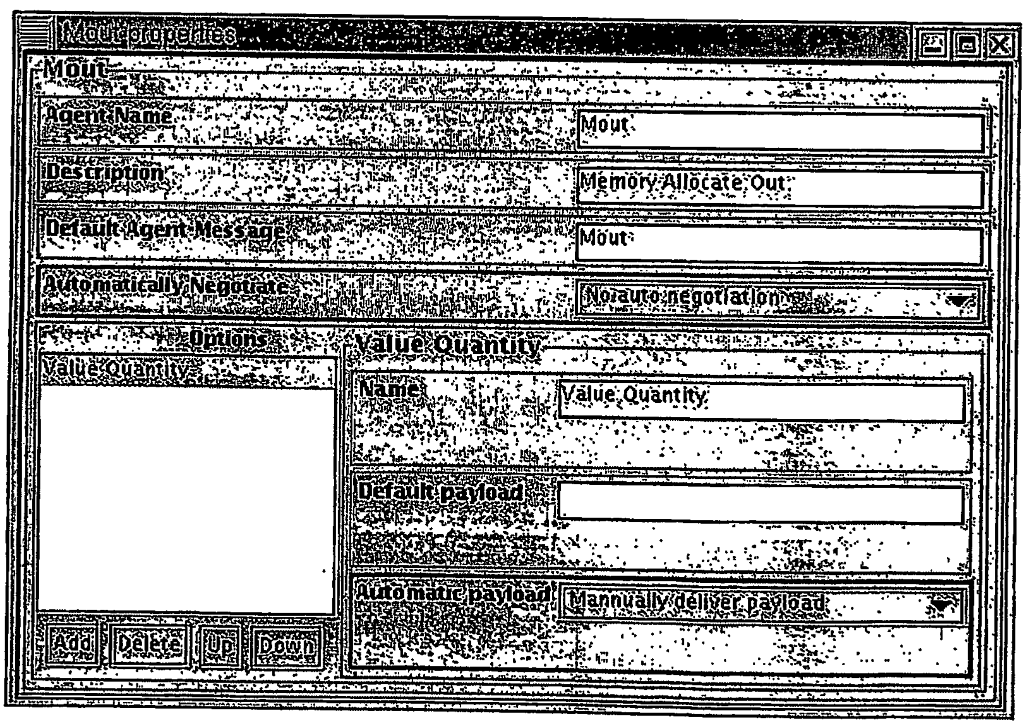

An entity determines a manner of providing a respective service
400
The entity defines a component encapsulating the provision of the service using the entity station 5
410
Component, input and output specifications are stored in the database 16
420
The entity accesses the base station 1 using the entity station 5
430
The component, input and output specifications are transferred to the base staiton 1
440
The specifications are stored in the database 11
450

Fig. 7



Example 1 properties
Example 1
Component Name
Example 1
Component Description
View Description
Component Author
Dave
Label X Position
10
Label Y Position
10
Address
192.168.2.14
Port Number
5000


Fig. 8



Mout properties
Mout
Agent Name
Mout
Description
Memory Allocate Out
Default Agent Message
Mout
Automatically Negotiate
No auto negotiation
Options
Value Quantity
Value Quantity
Name
Value Quantity
Default payload
Automatic payload
Mannually deliver payload
Add
Delete
Up
Down


Fig. 9

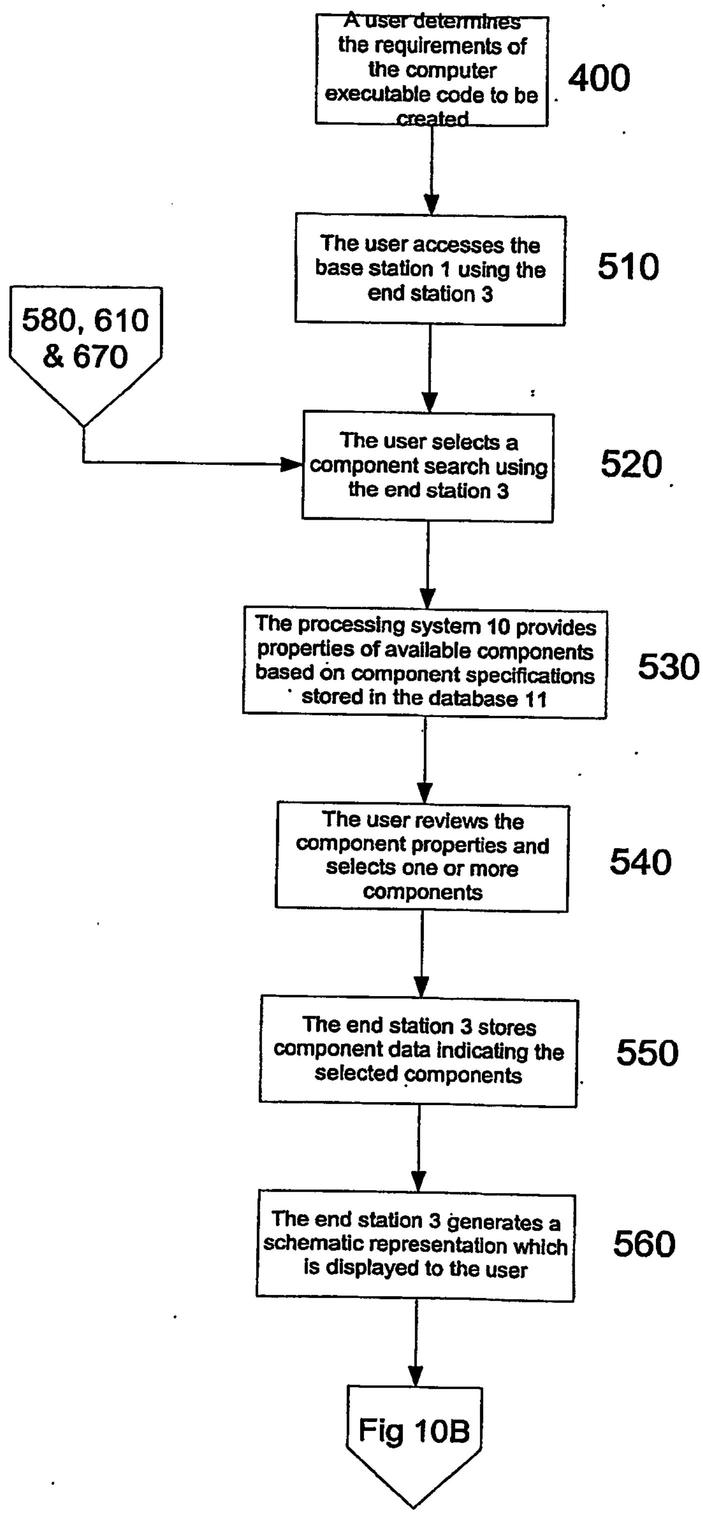

A user determines the requirements of the computer executable code to be created
400
The user accesses the base station 1 using the end station 3
510
580, 610 & 670
The user selects a component search using the end station 3
520
The processing system 10 provides properties of available components based on component specifications stored in the database 11
530
The user reviews the component properties and selects one or more components
540
The end station 3 stores component data indicating the selected components
550
The end station 3 generates a schematic representation which is displayed to the user
560
Fig 10B

Fig. 10A

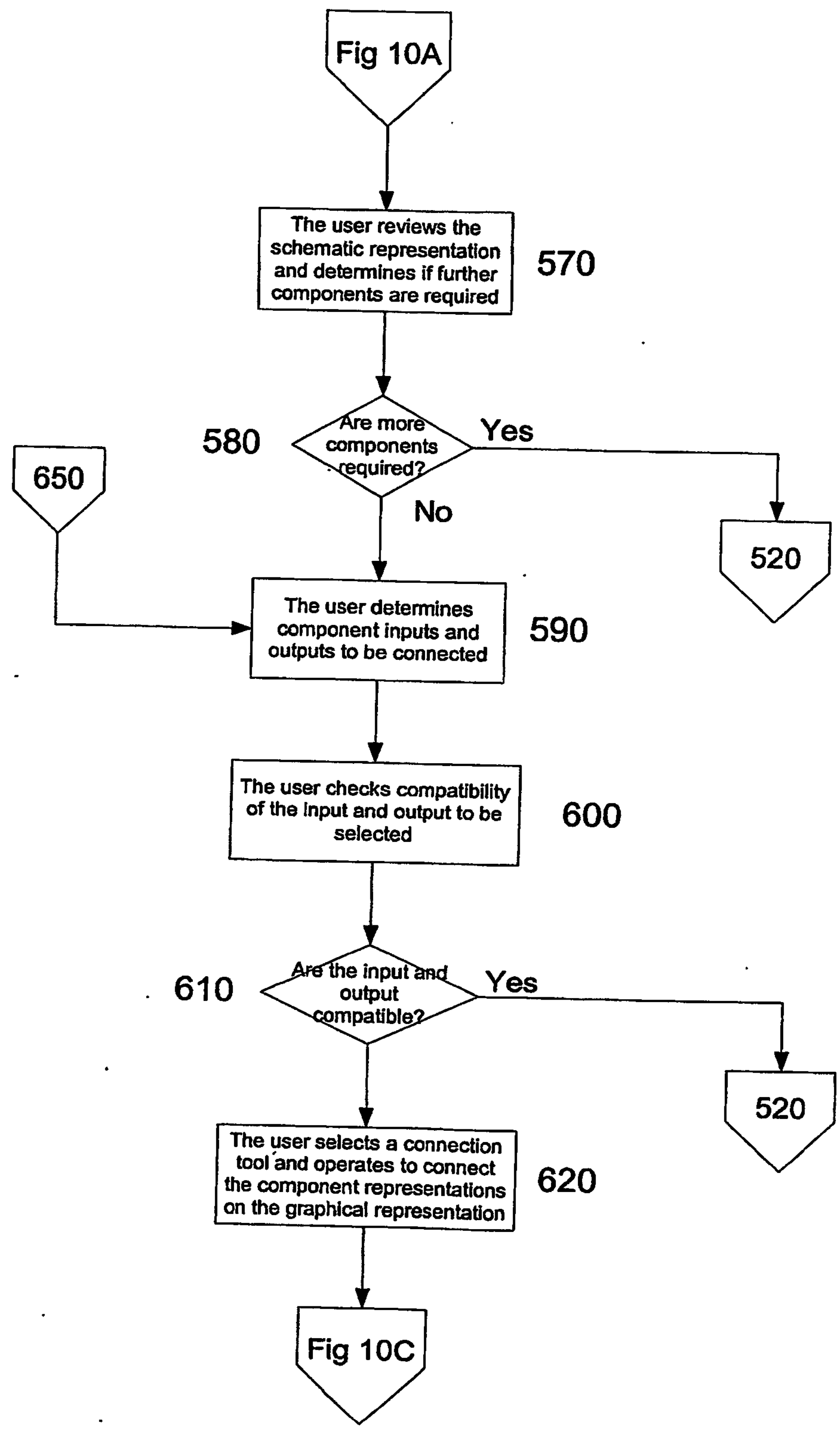


Fig 10A
The user reviews the schematic representation and determines if further components are required
570
Are more components required?
580
Yes
No
650
520
The user determines component inputs and outputs to be connected
590
The user checks compatibility of the input and output to be selected
600
Are the input and output compatible?
610
Yes
520
The user selects a connection tool and operates to connect the component representations on the graphical representation
620
Fig 10C

**Fig. 10B**

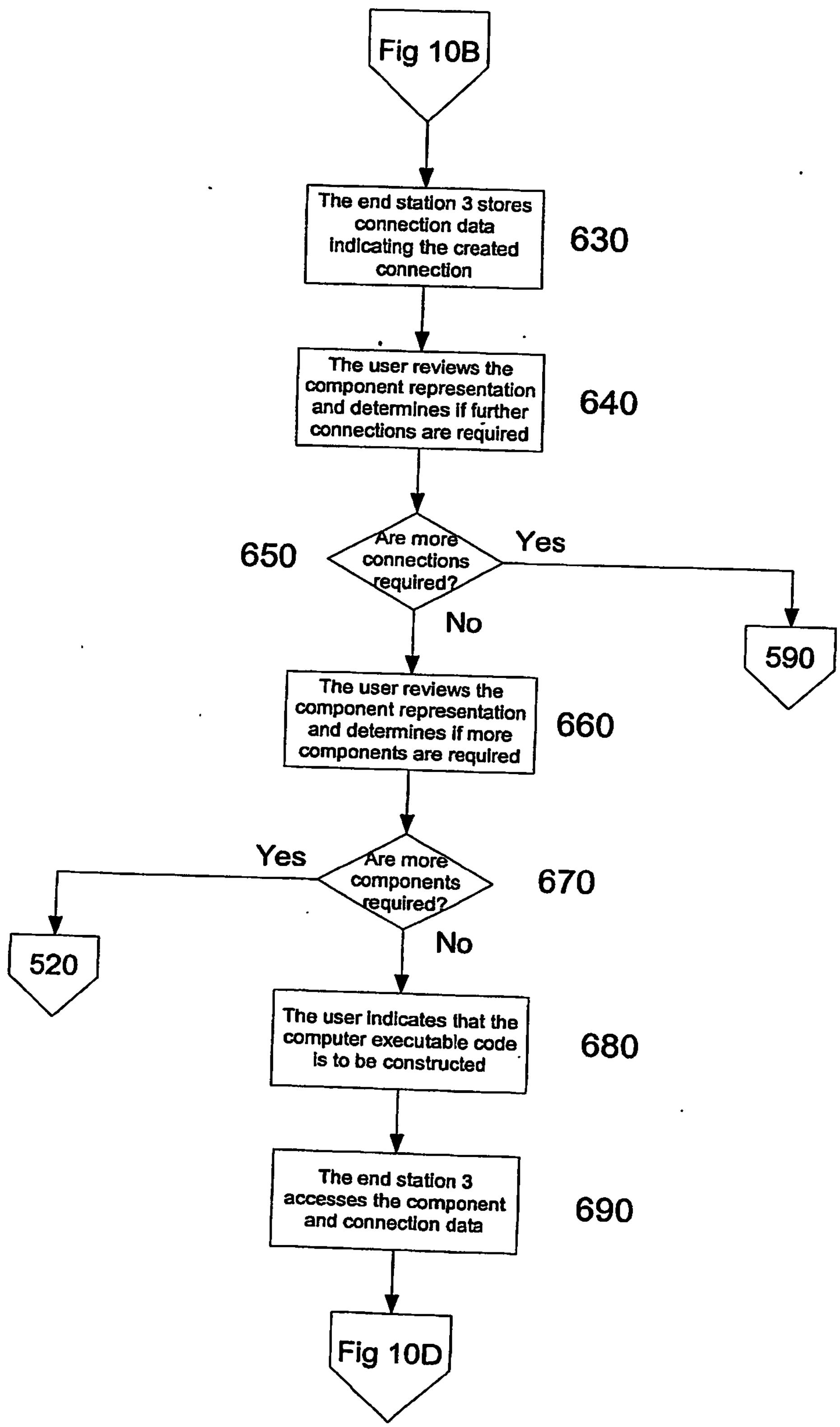


Fig 10B
The end station 3 stores connection data indicating the created connection
630
The user reviews the component representation and determines if further connections are required
640
650
Are more connections required?
Yes
No
590
The user reviews the component representation and determines if more components are required
660
Yes
Are more components required?
670
No
520
The user indicates that the computer executable code is to be constructed
680
The end station 3 accesses the component and connection data
690
Fig 10D

Fig. 10C

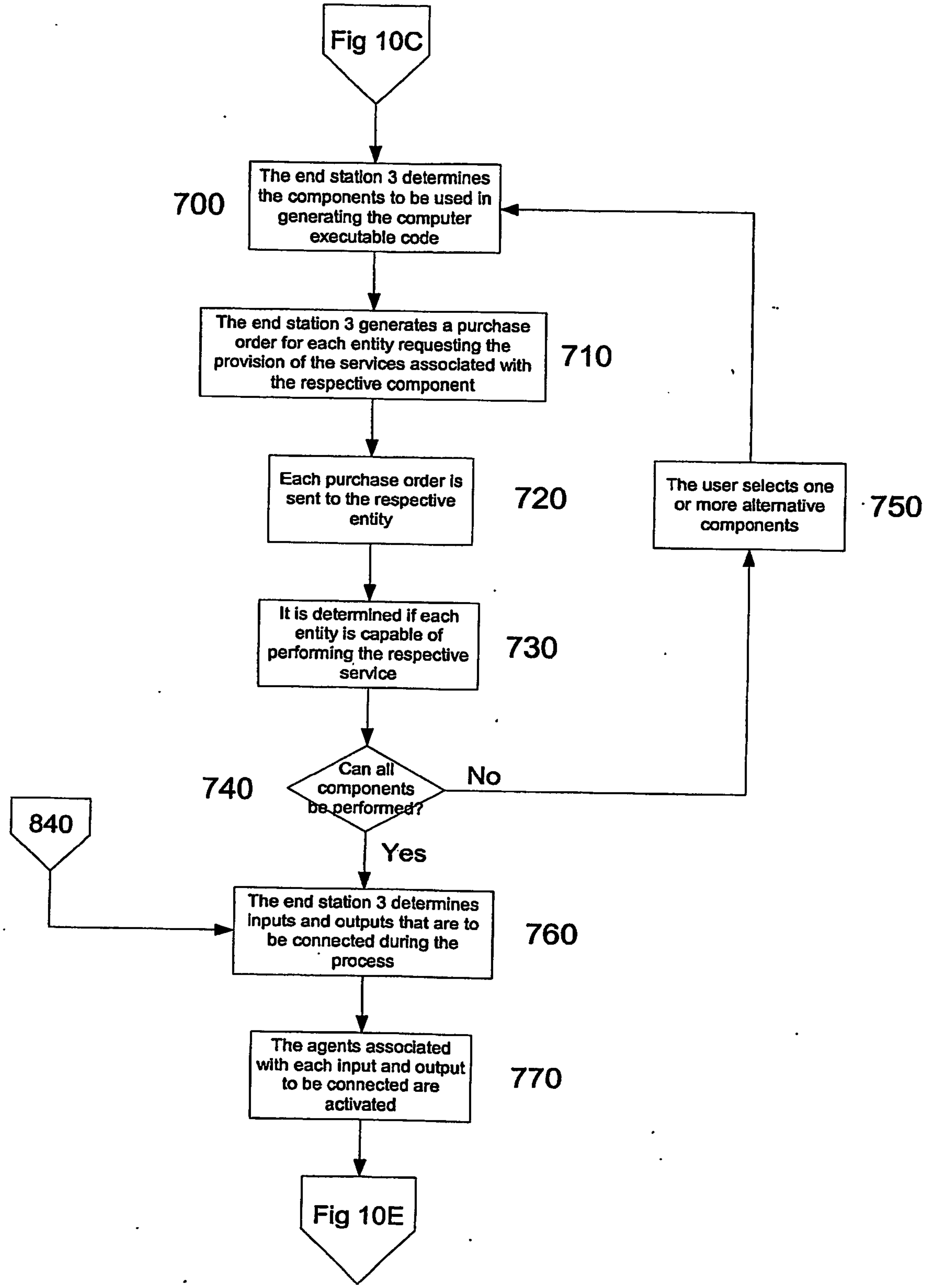


Fig 10C
The end station 3 determines the components to be used in generating the computer executable code
700
The end station 3 generates a purchase order for each entity requesting the provision of the services associated with the respective component
710
Each purchase order is sent to the respective entity
720
The user selects one or more alternative components
750
It is determined if each entity is capable of performing the respective service
730
Can all components be performed?
740
No
Yes
840
The end station 3 determines inputs and outputs that are to be connected during the process
760
The agents associated with each input and output to be connected are activated
770
Fig 10E

Fig. 10D

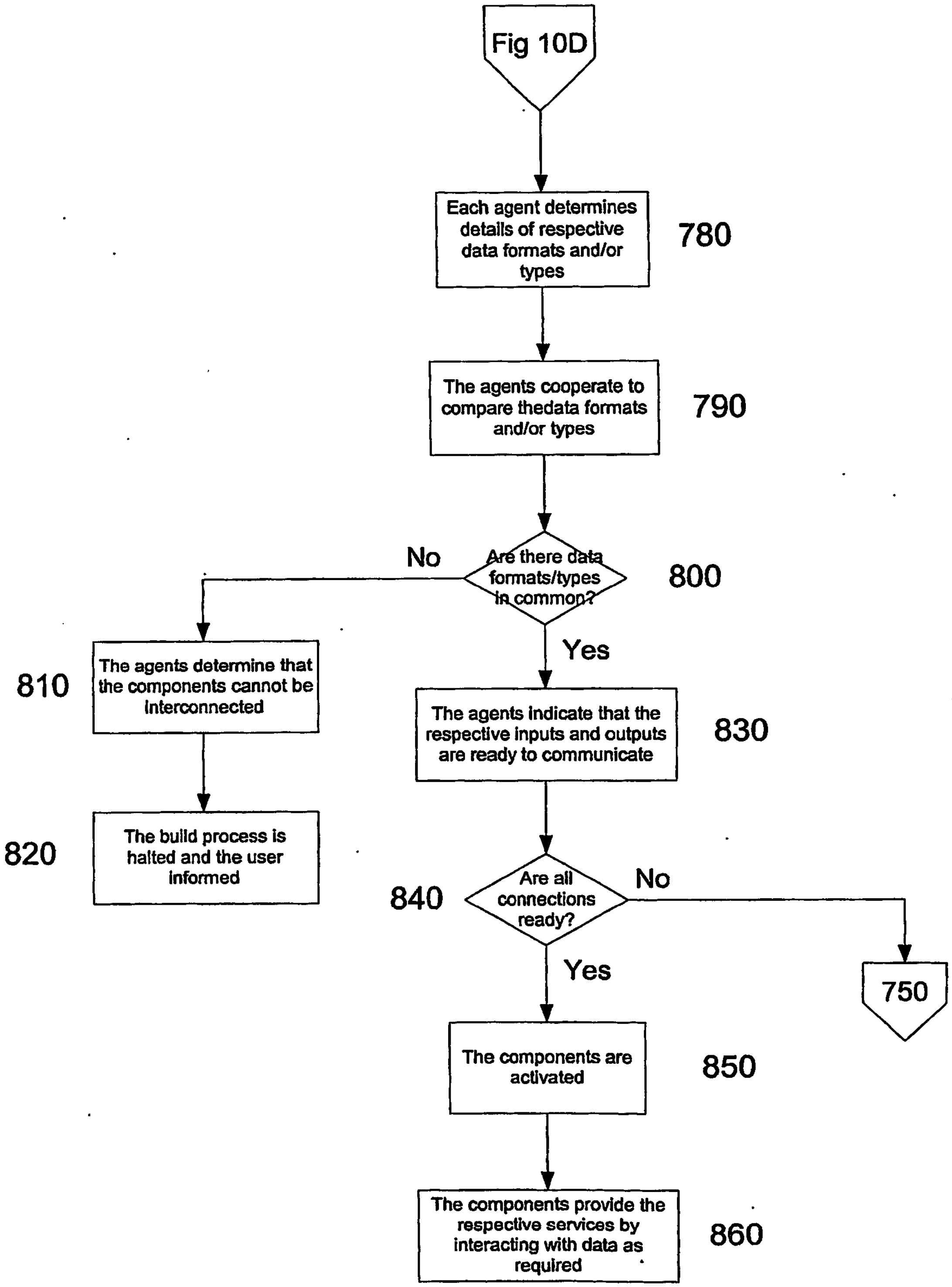


Fig 10D
Each agent determines details of respective data formats and/or types
780
The agents cooperate to compare thedata formats and/or types
790
Are there data formats/types in common?
800
No
Yes
810
The agents determine that the components cannot be interconnected
820
The build process is halted and the user informed
The agents indicate that the respective inputs and outputs are ready to communicate
830
840
Are all connections ready?
No
Yes
750
The components are activated
850
The components provide the respective services by interacting with data as required
860

Fig. 10E

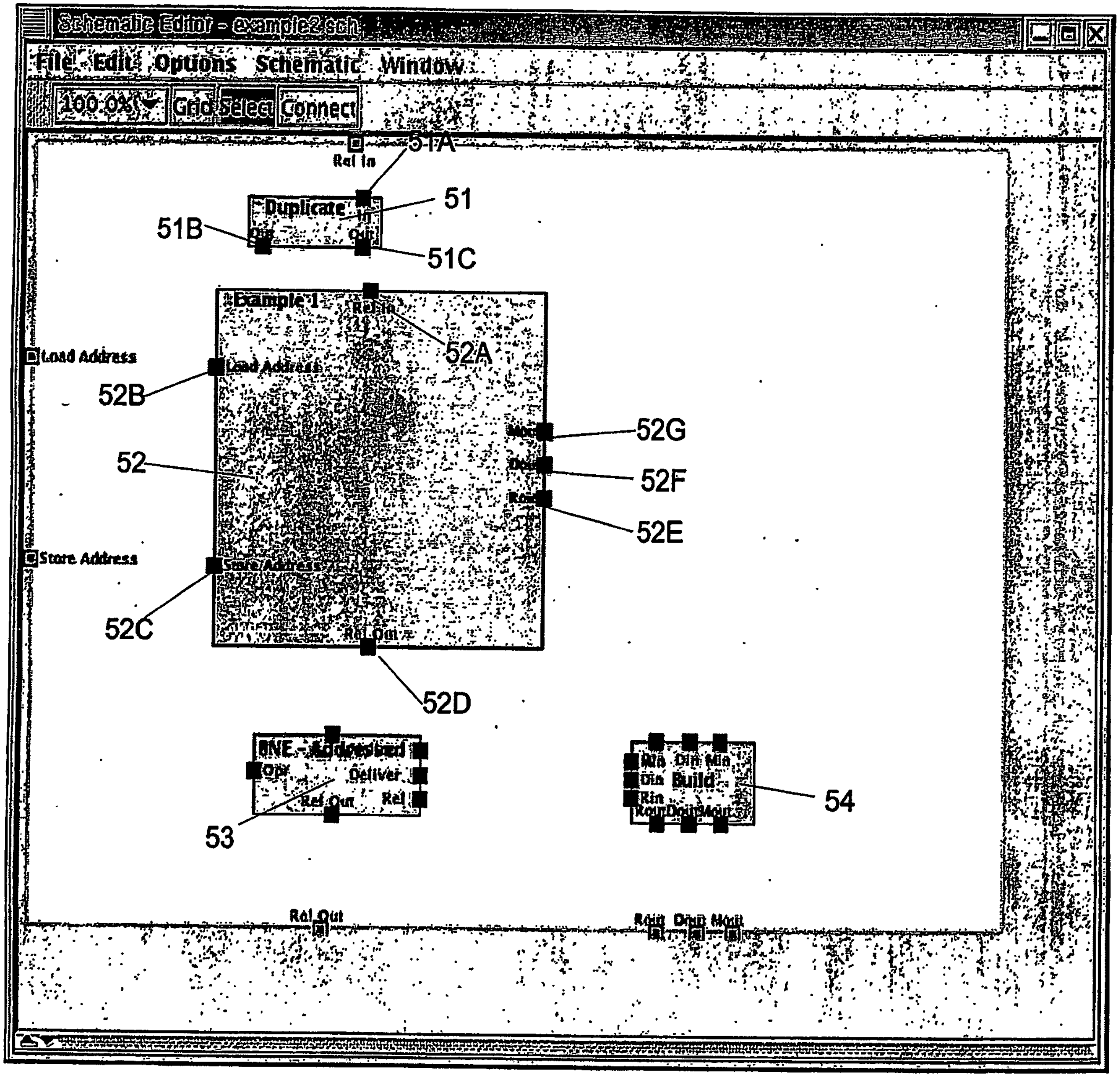

Schematic Editor - example2.sch
File Edit Options Schematic Window
100.0%
Grid
Select
Connect
Rel In
51A
51
Duplicate
51B
51C
Example 1
52A
Load Address
52B
52
52G
52F
52E
Store Address
52C
Rel Out
52D
Deliver
Rel
53
Build
54
Rel Out

Fig. 11

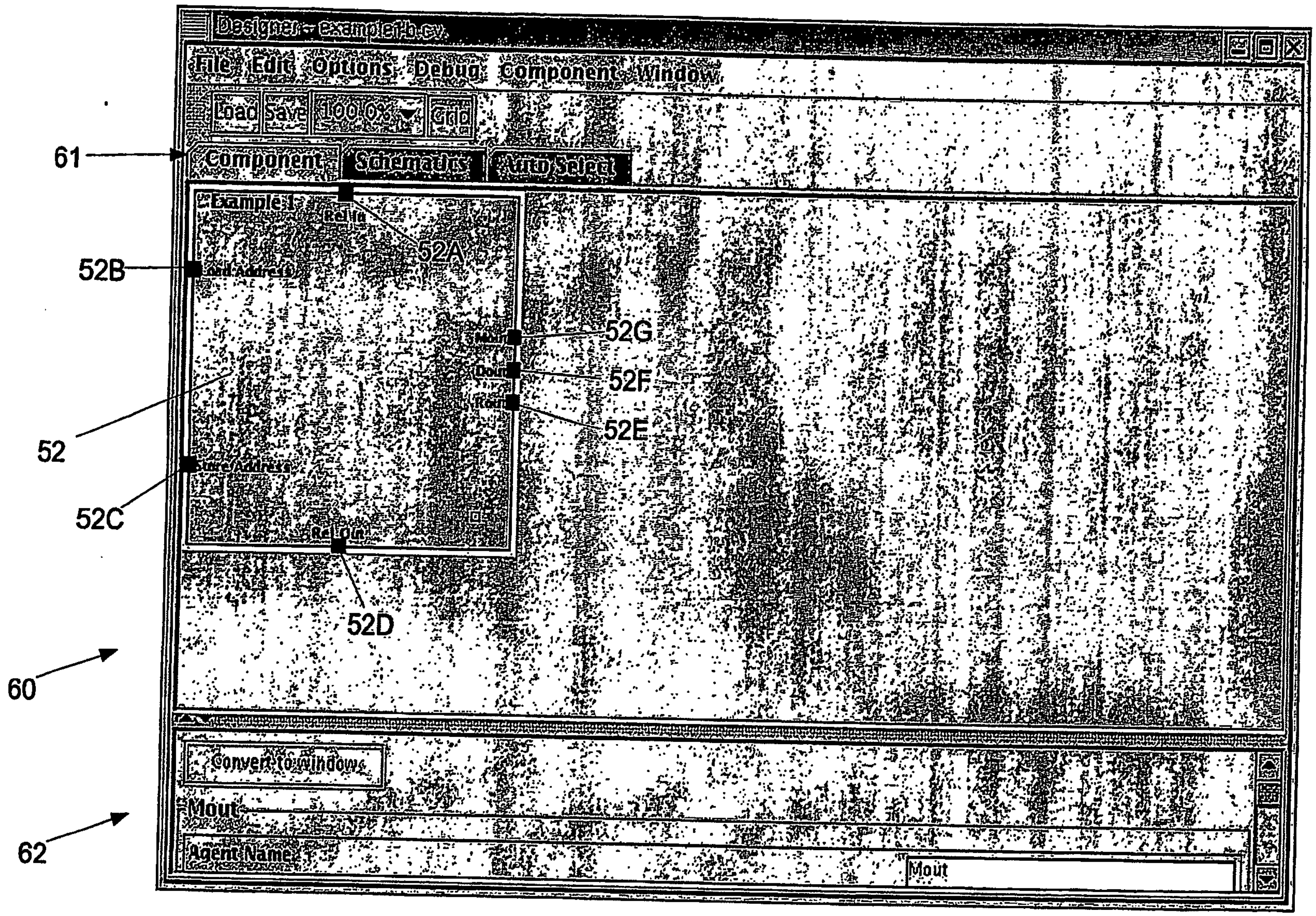

Designer - example1b.cv
File Edit Options Debug Component Window
Load Save 100.0% Grid
Component Schematics Auto Select
Example 1
Convert to windows
Mout
Agent Name
Mout
61
52A
52B
52G
52F
52E
52
52C
52D
60
62

Fig. 12

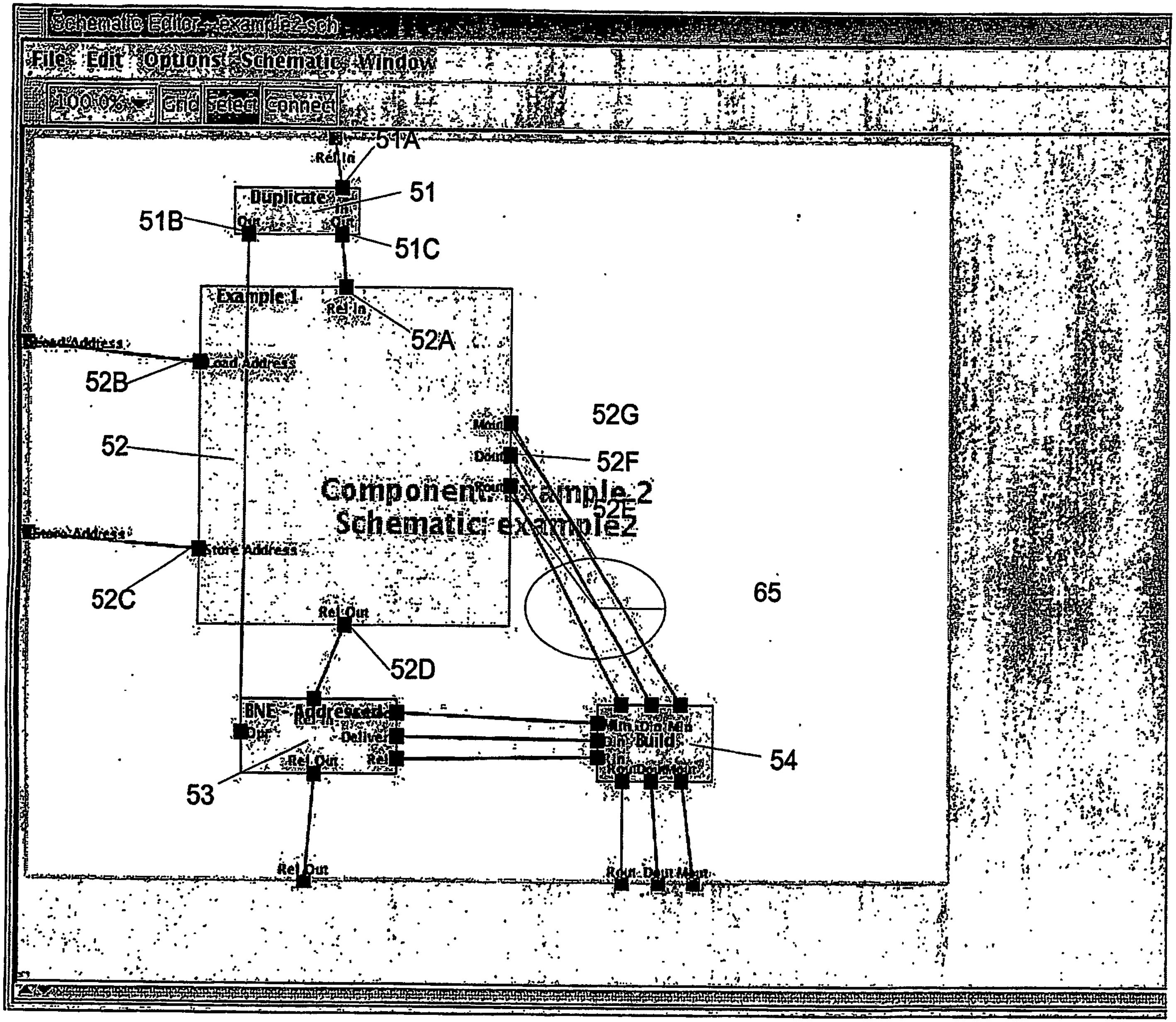
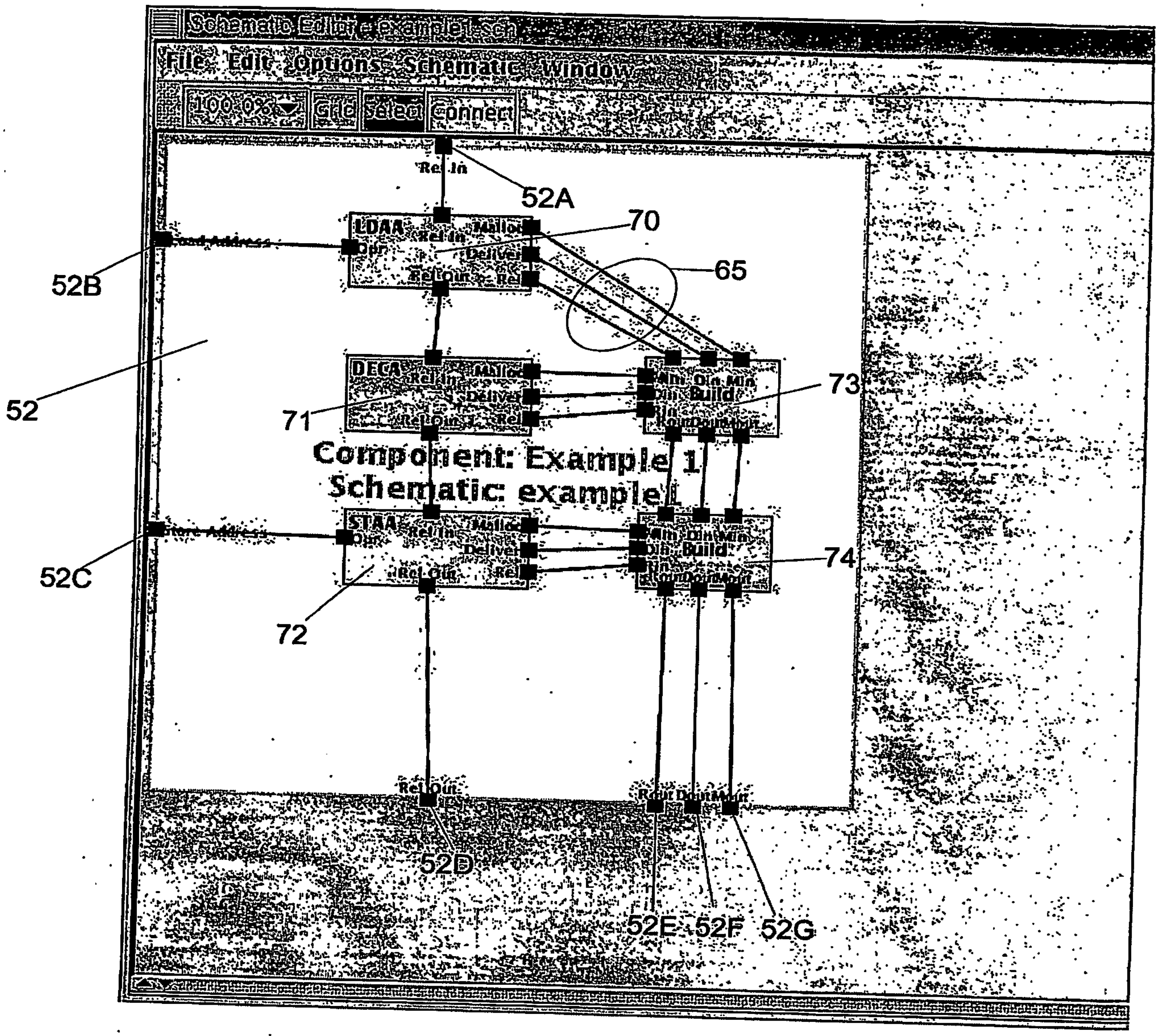

Schematic Editor - example2.sch
File Edit Options Schematic Window
100.0%
Grid
Select
Connect
Rel In
51A
Duplicate
51
51B
Out
Out
51C
Example 1
Rel In
52A
Load Address
Load Address
52B
52
52G
52F
Dout
Component: Example 2
Schematic: example2
52E
Store Address
Store Address
52C
65
Rel Out
52D
BNE - Addressed
Rel In
Out
Deliver
Rel
Rel Out
53
Build
54
Rel Out
Dout

Fig. 13



File Edit Options Schematic Window
100.0% Grid Select Connect
Rel In
52A
70
65
52B
LDAA
Rel In
Malloc
Opr
Deliver
Rel Out
Rel
52
DECA
Rel In
Malloc
Deliver
Rel Out
Rel
71
Min
Din
Rin
Build
Rout
Dout
Mout
73
Component: Example 1
Schematic: example1
STAA
Rel In
Malloc
Opr
Deliver
Rel Out
Rel
52C
Store Address
Build
74
72
Rel Out
52D
Rout
Dout
Mout
52E
52F
52G


Fig. 14

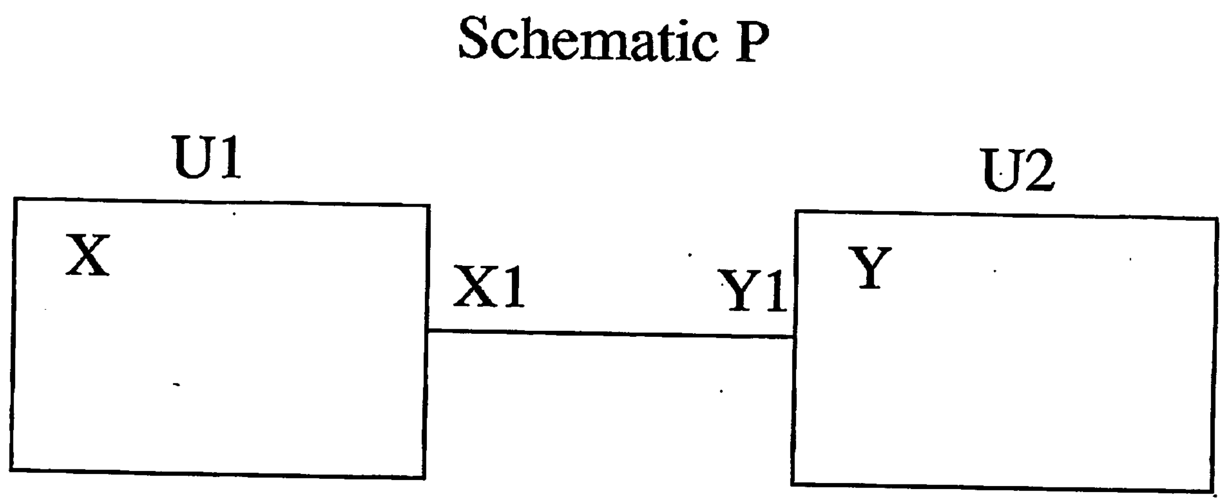

Schematic P
U1
U2
X
X1
Y1
Y

**Fig. 15**

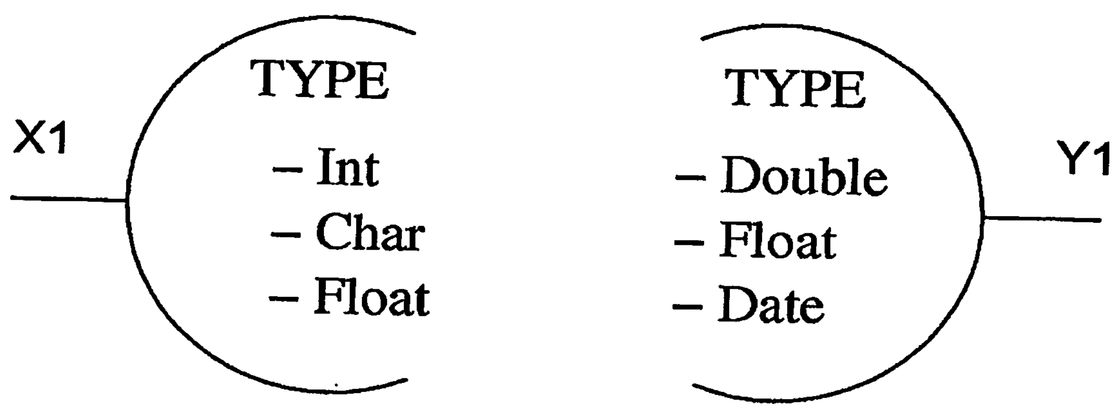

X1
TYPE
– Int
– Char
– Float
TYPE
– Double
– Float
– Date
Y1

**Fig. 16**

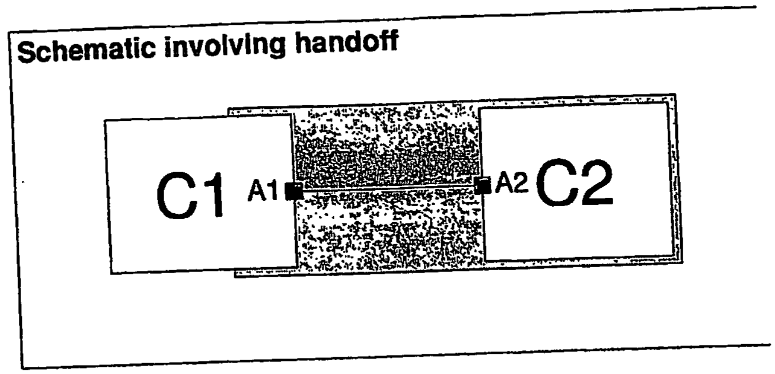



Fig. 17A

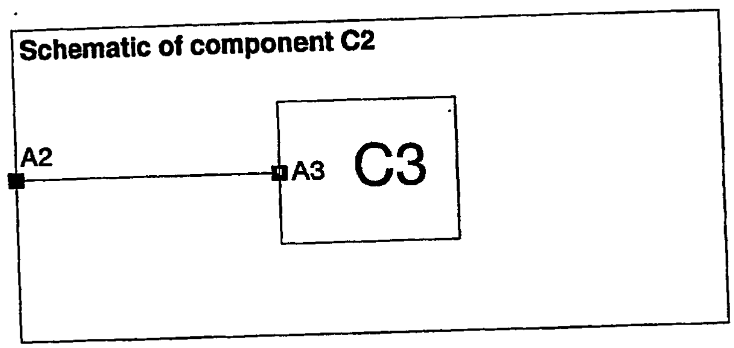



Fig. 17B

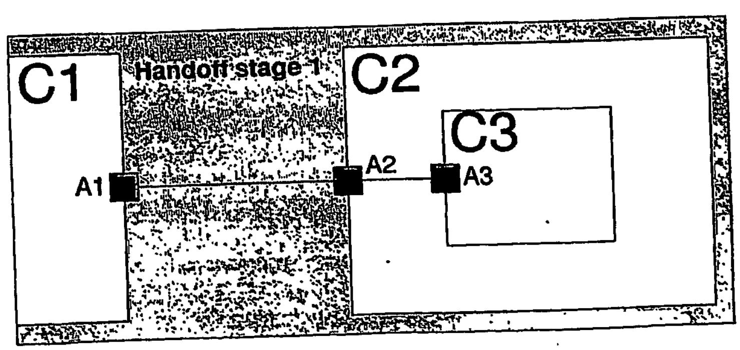



Fig. 17C

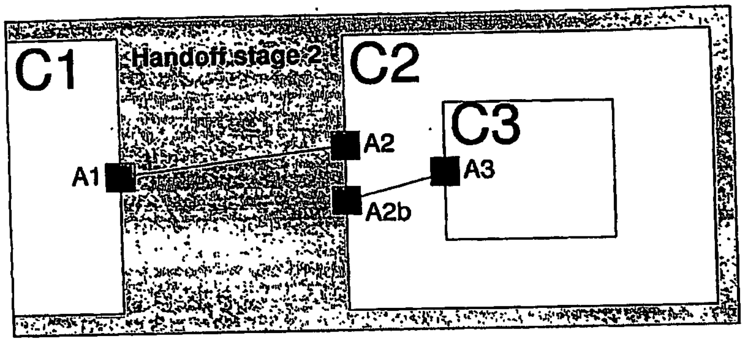



**Fig. 17D**

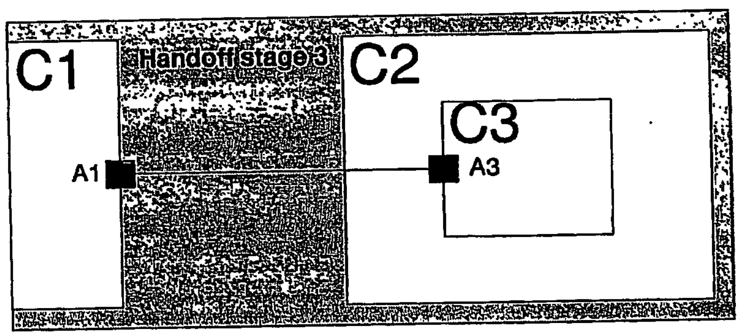



**Fig. 17E**

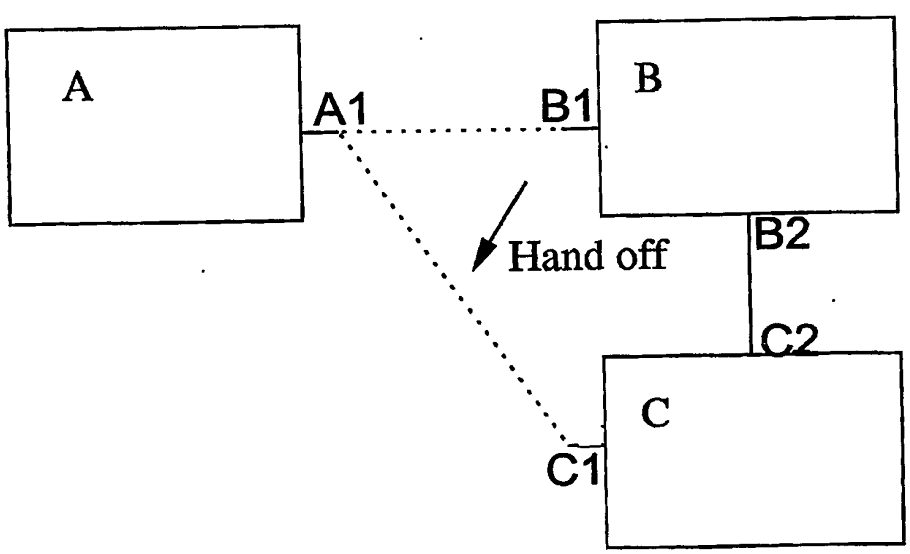



**Fig. 18**

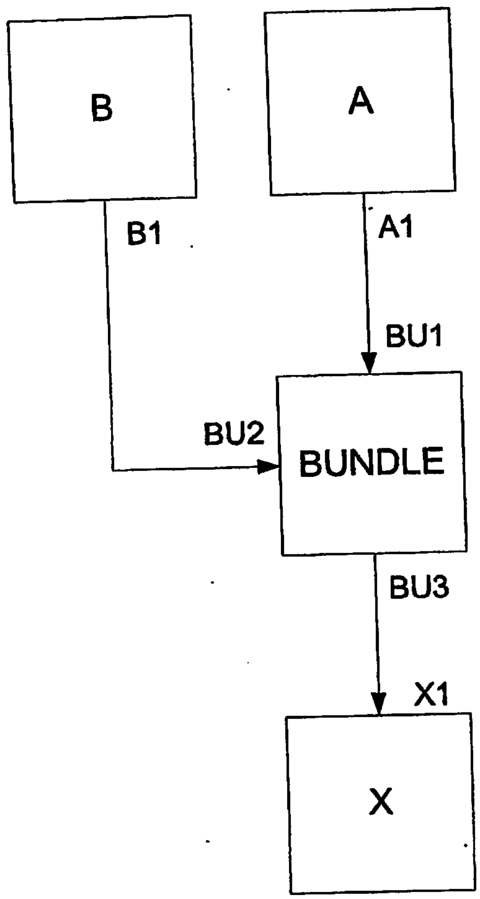



Fig. 19A

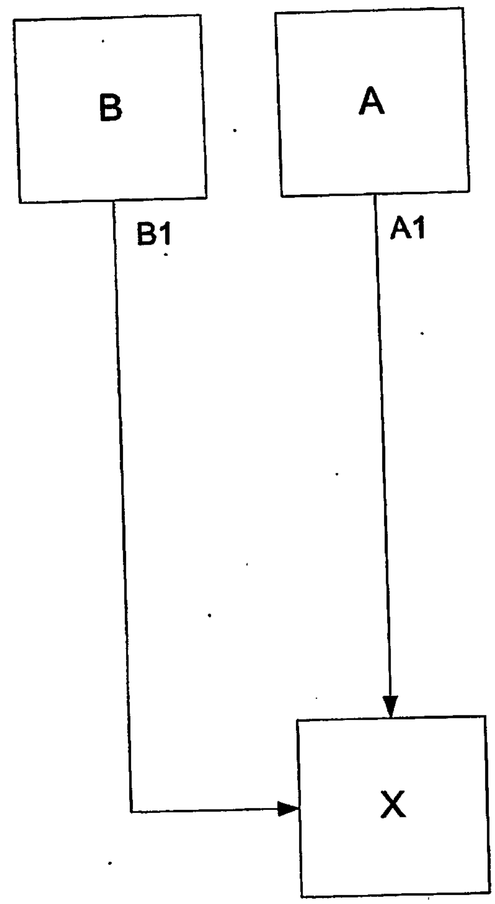



Fig. 19B

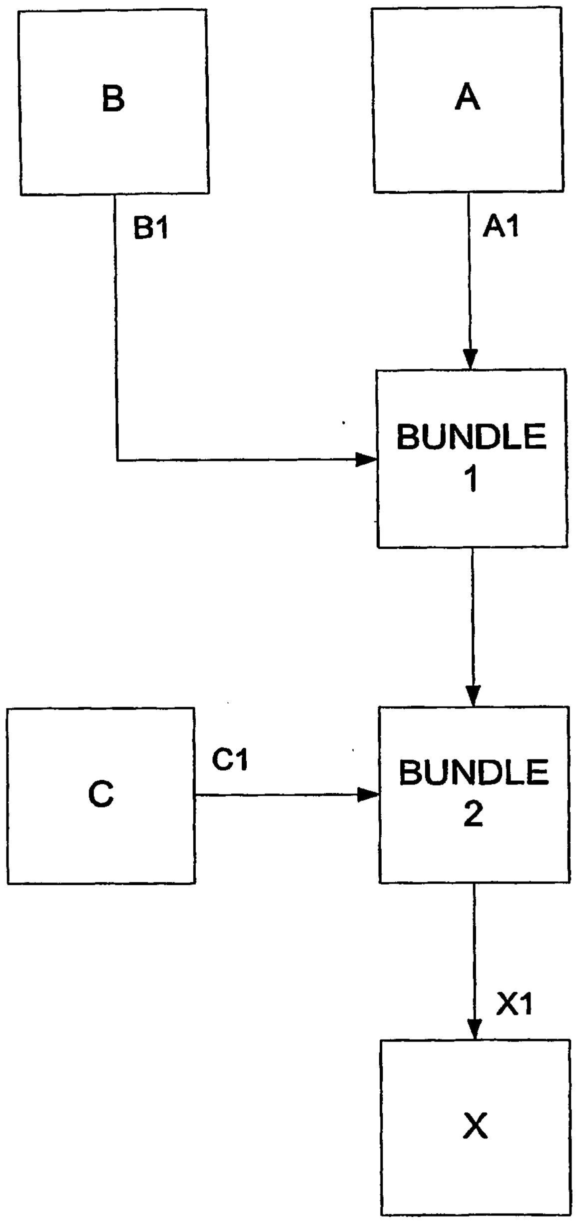



Fig. 20

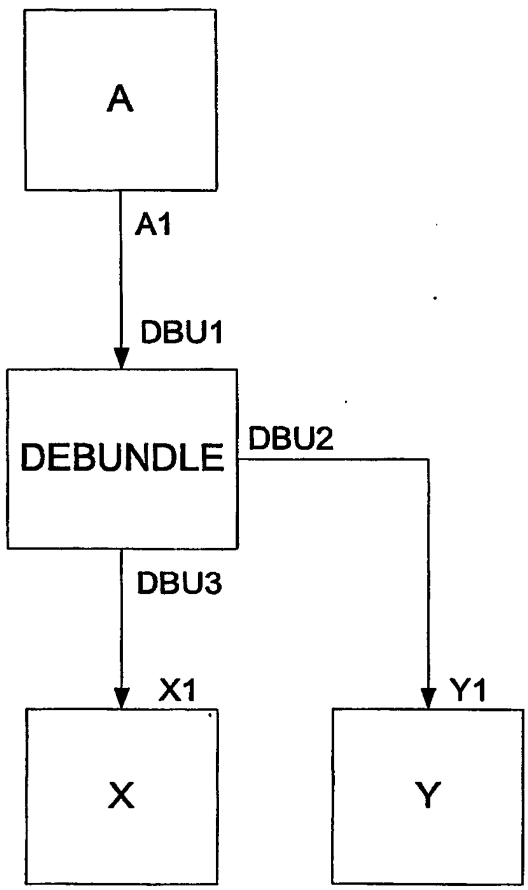



**Fig. 21**

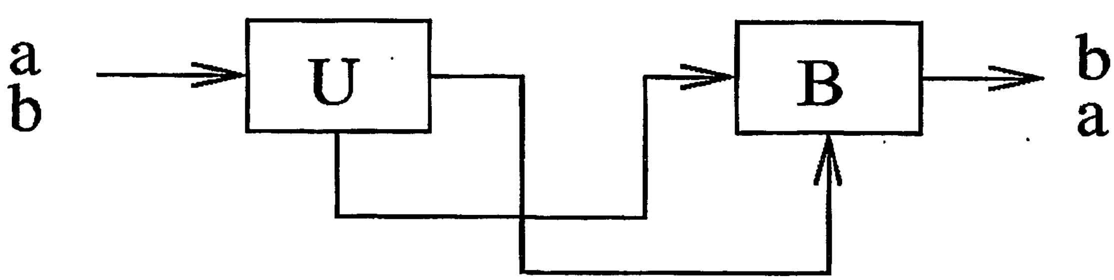



**Fig. 23**

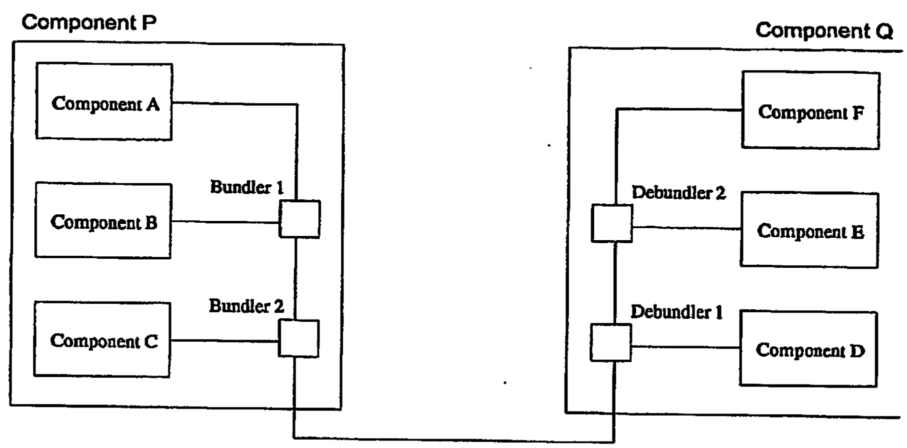



Fig. 22A

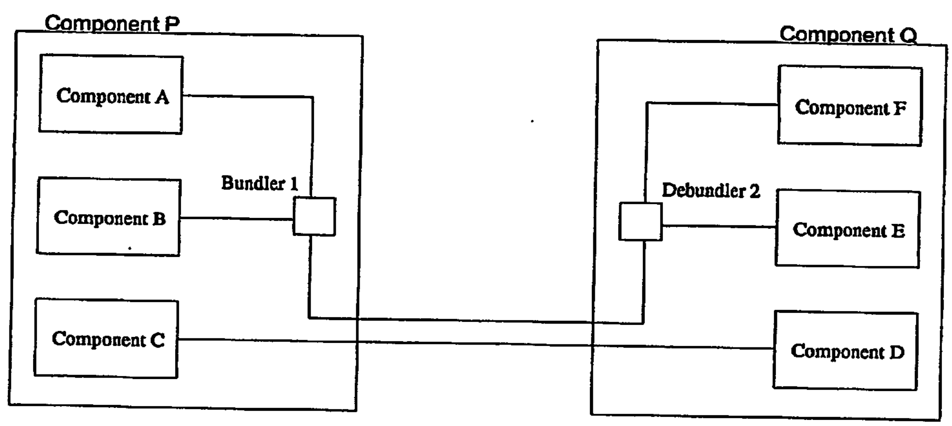



Fig. 22B

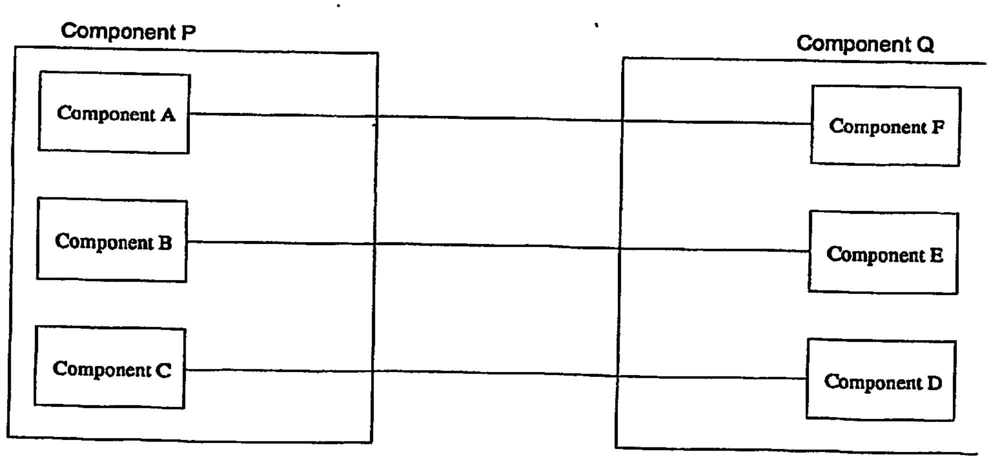



Fig. 22C

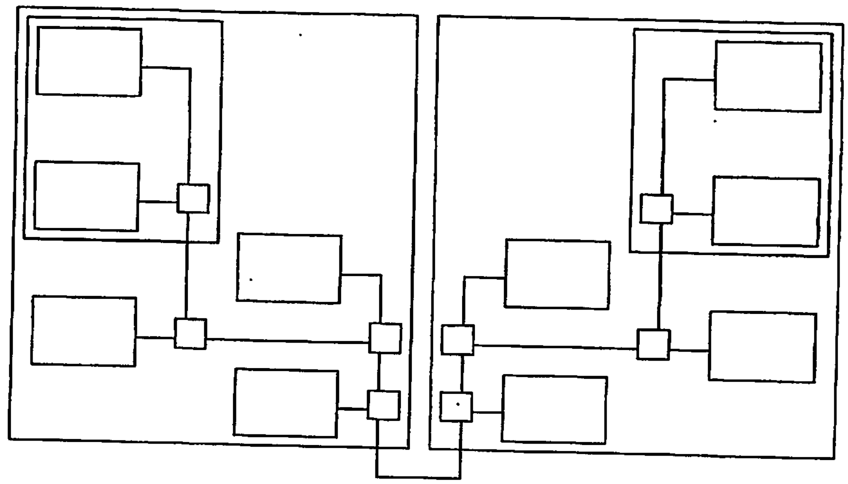


Fig. 22D

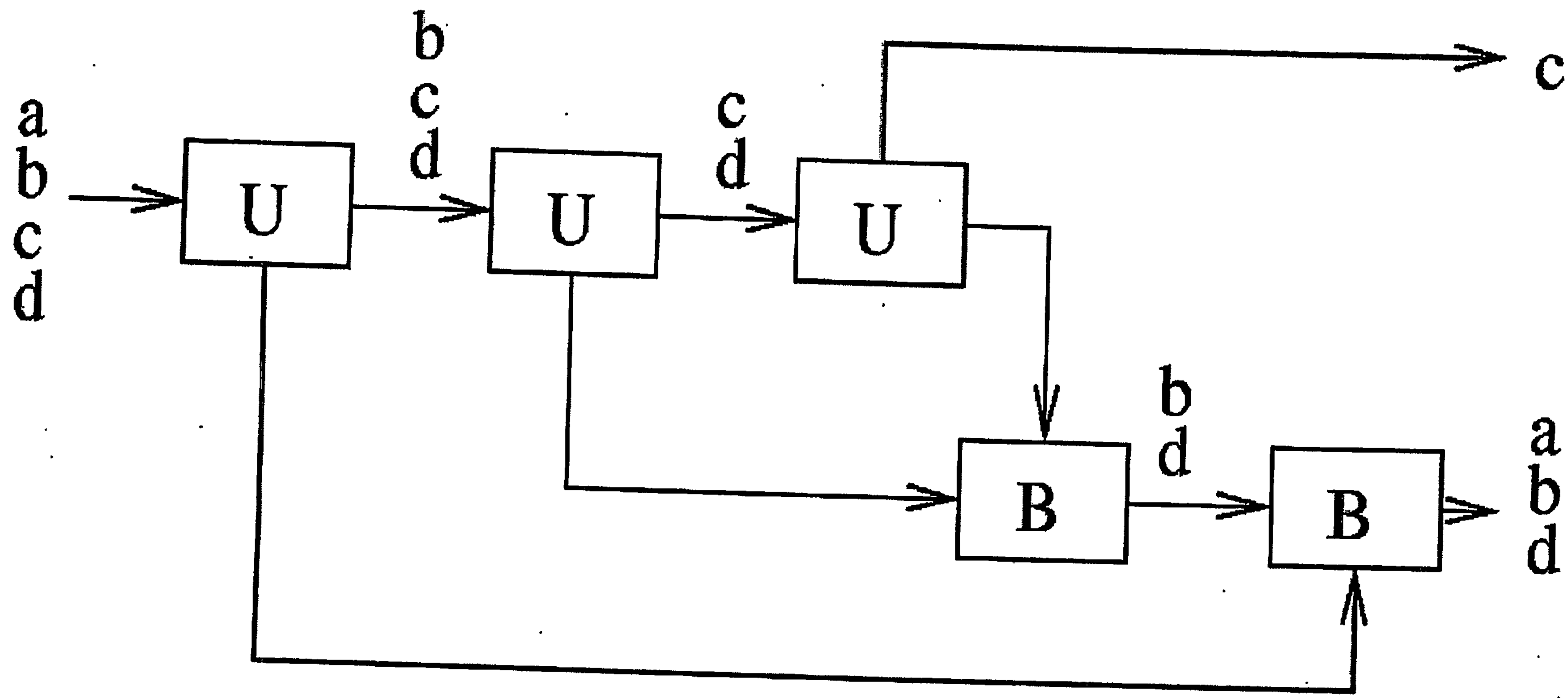
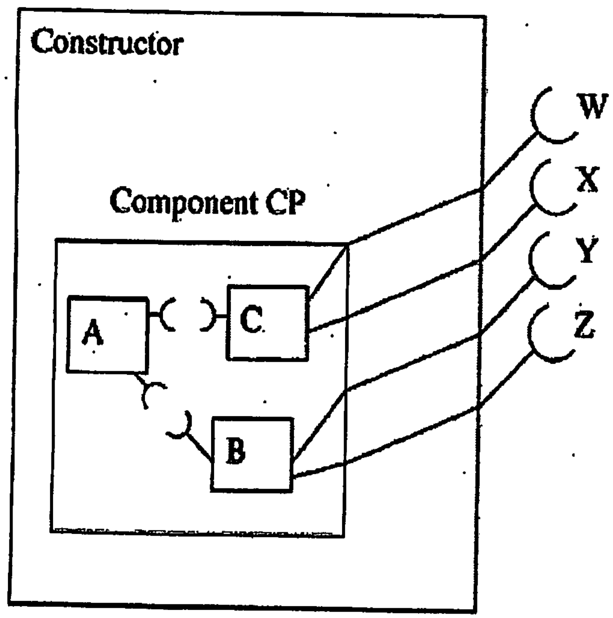
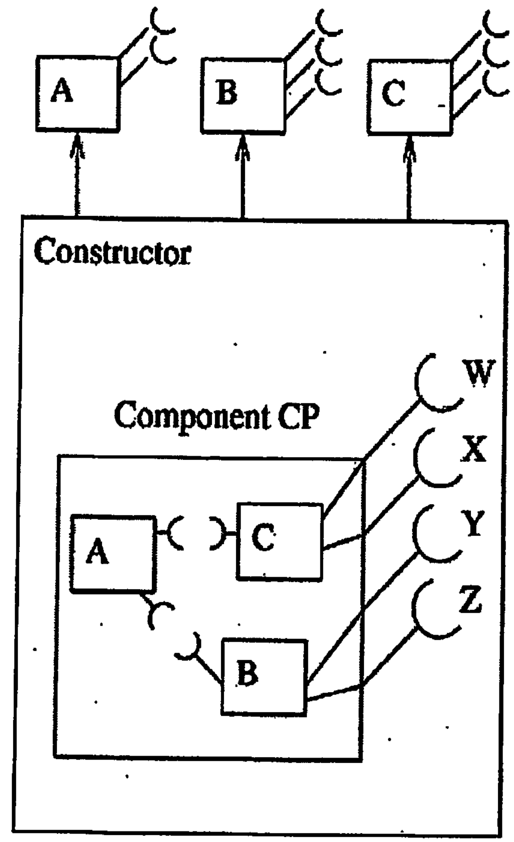



Fig. 24



Constructor
Component CP
A
C
B
W
X
Y
Z


Fig. 25



A
B
C
Constructor
Component CP
A
C
B
W
X
Y
Z


Fig. 26

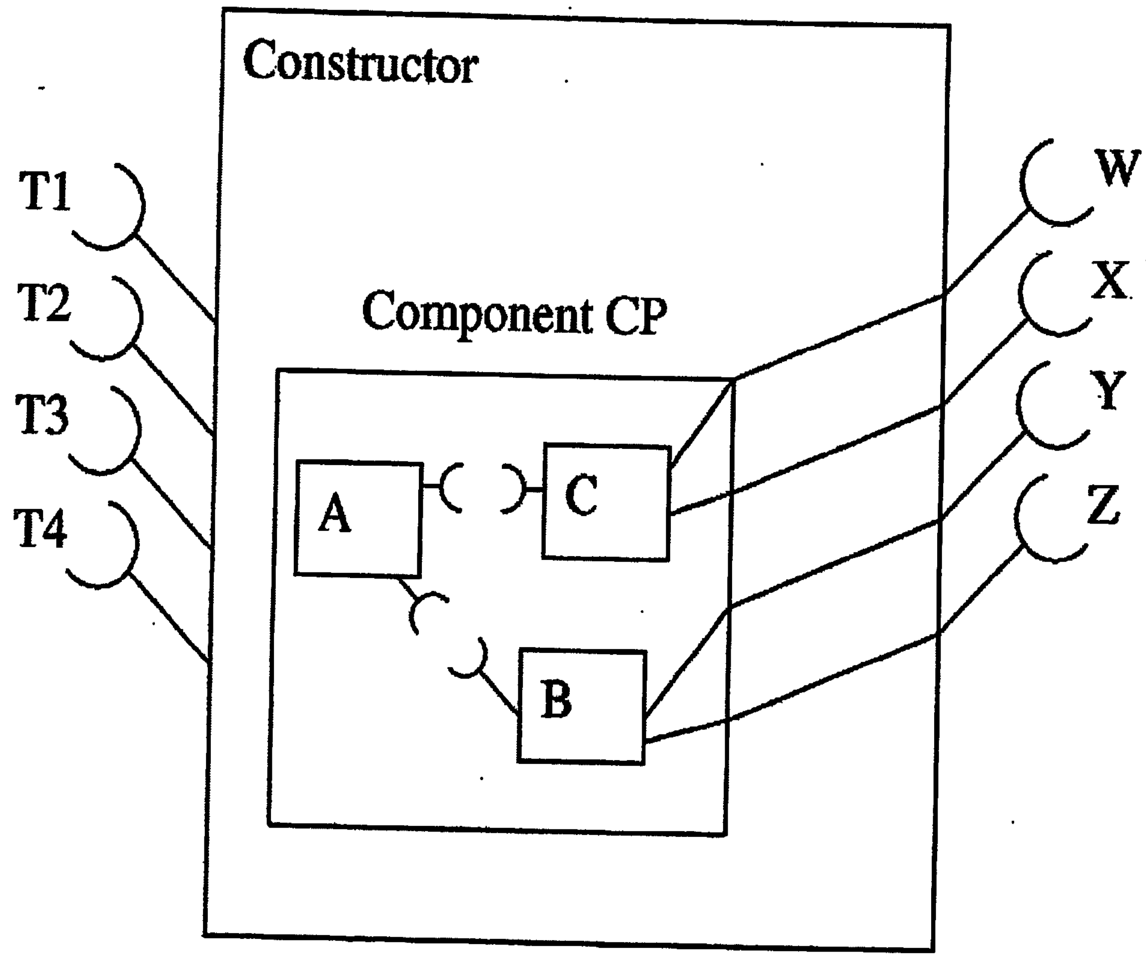



Fig. 27

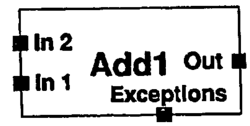



Fig. 28

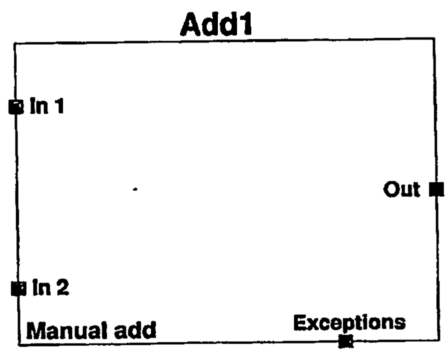



Fig. 29

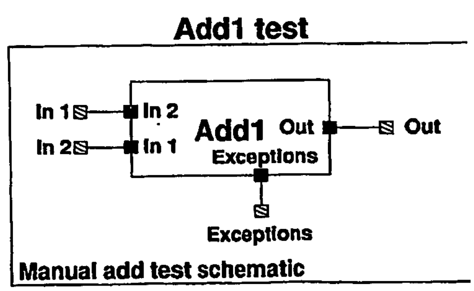



Fig. 30

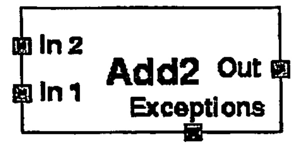



**Fig. 31**

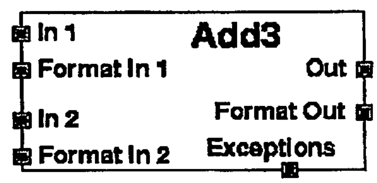



**Fig. 32**

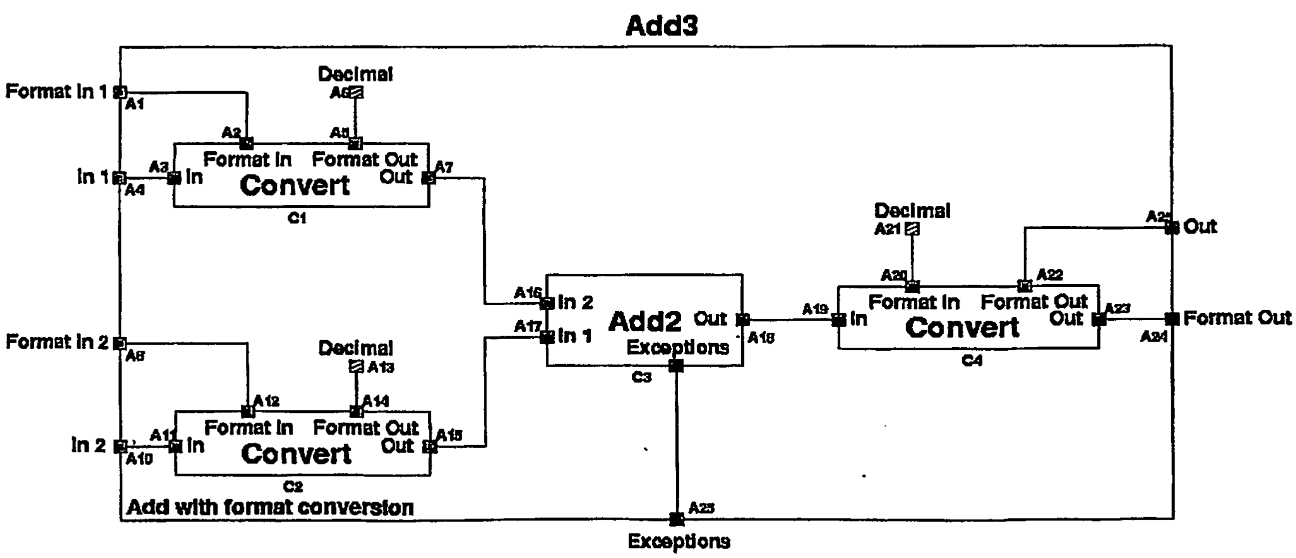



**Fig. 33**

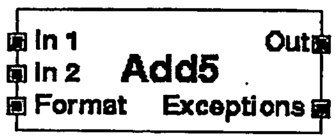



Fig. 34

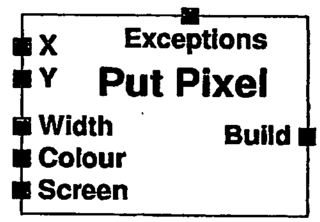



Fig. 36

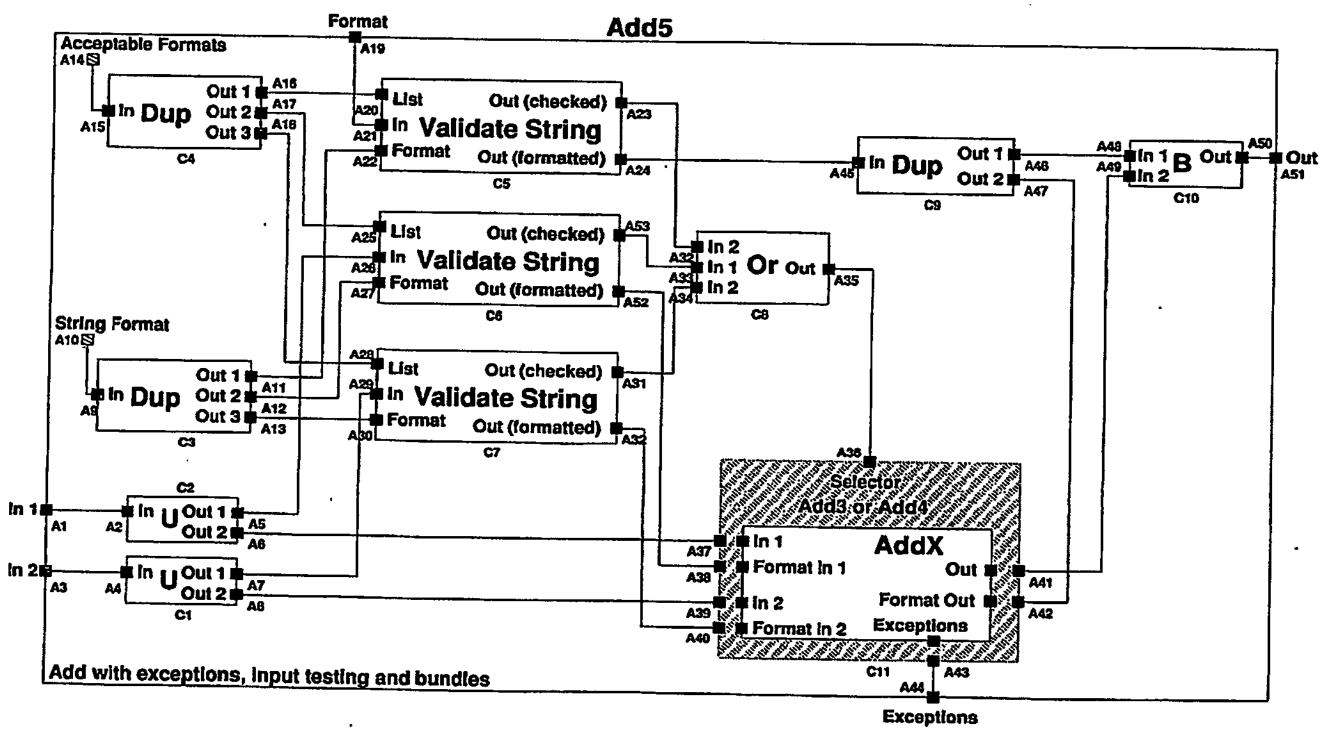
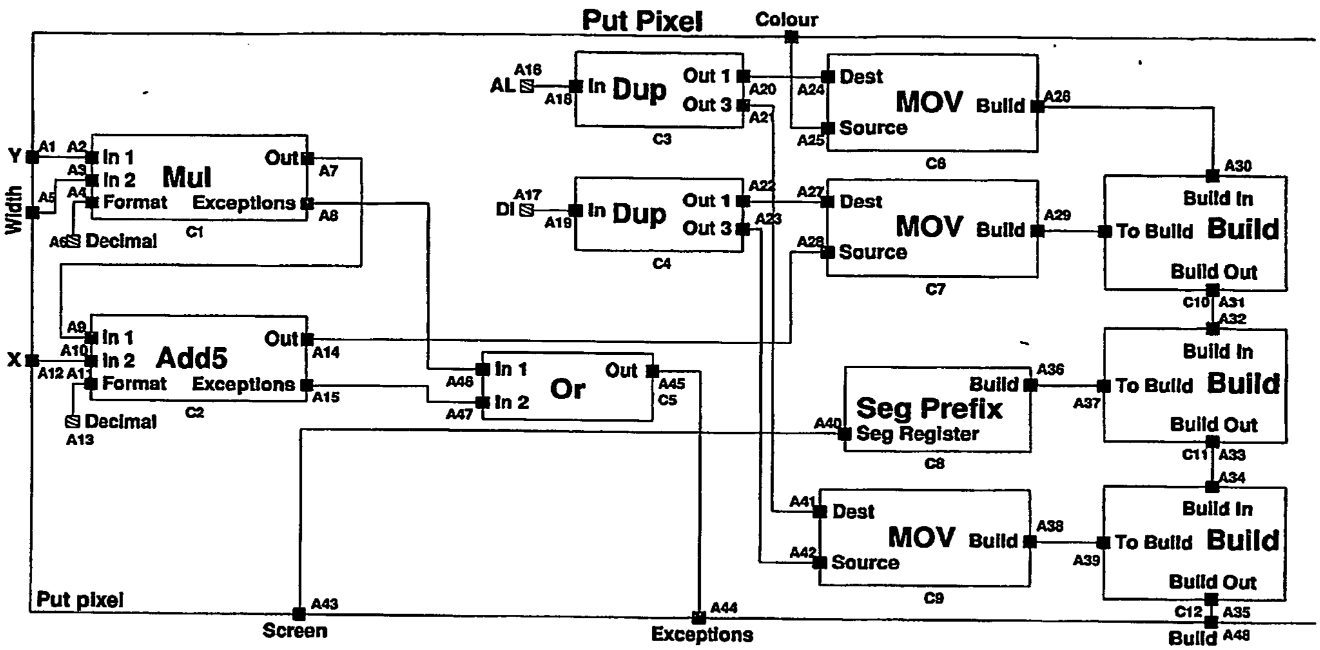
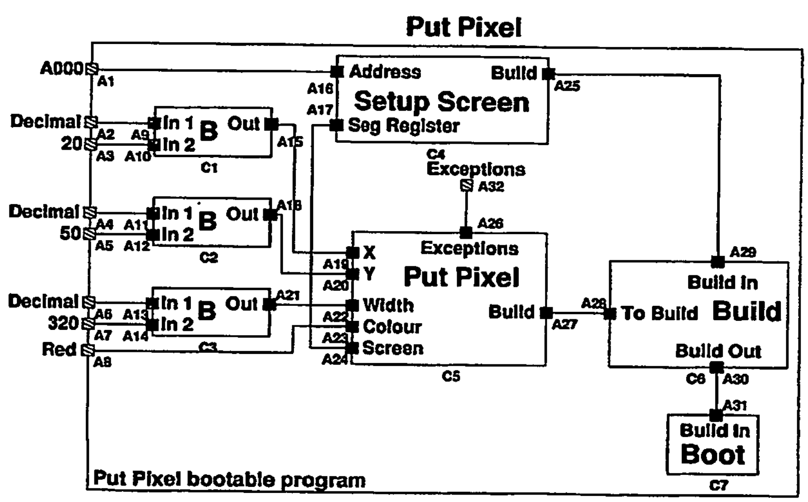



Fig. 35



Put Pixel
Colour
AL
DI
Y
X
Width
In
Dup
Out 1
Out 3
MOV
Dest
Source
Build
Build In
To Build
Build Out
In 1
In 2
Mul
Out
Format
Exceptions
Decimal
Add5
Or
Seg Prefix
Seg Register
Put pixel
Screen
Exceptions
Build


Fig. 37



Put Pixel
A000
Decimal
20
Decimal
50
Decimal
320
Red
In 1
In 2
B
Out
Address
Build
Setup Screen
Seg Register
Exceptions
X
Y
Width
Colour
Screen
Put Pixel
Build In
To Build
Build
Build Out
Boot
Put Pixel bootable program


Fig. 38

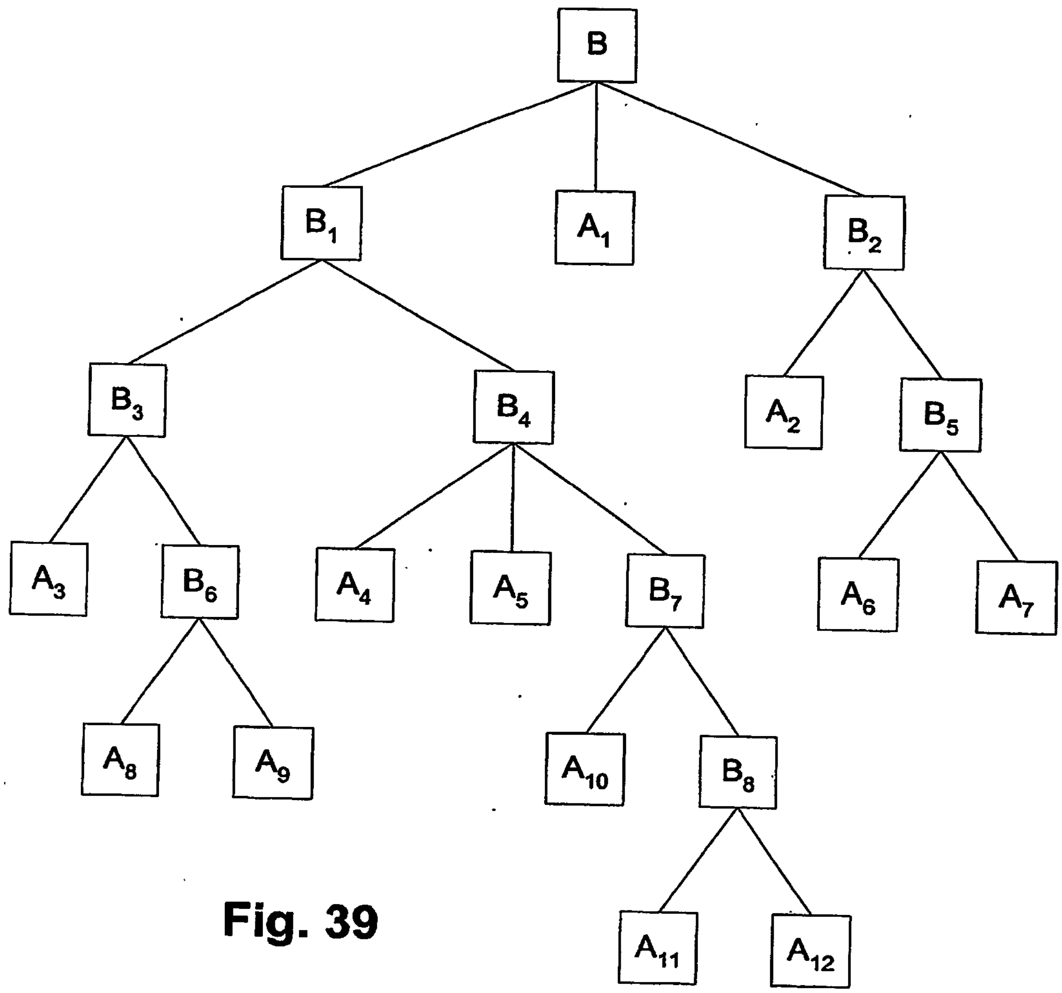

B
B1
A1
B2
B3
B4
A2
B5
A3
B6
A4
A5
B7
A6
A7
A8
A9
A10
B8
A11
A12

Fig. 39

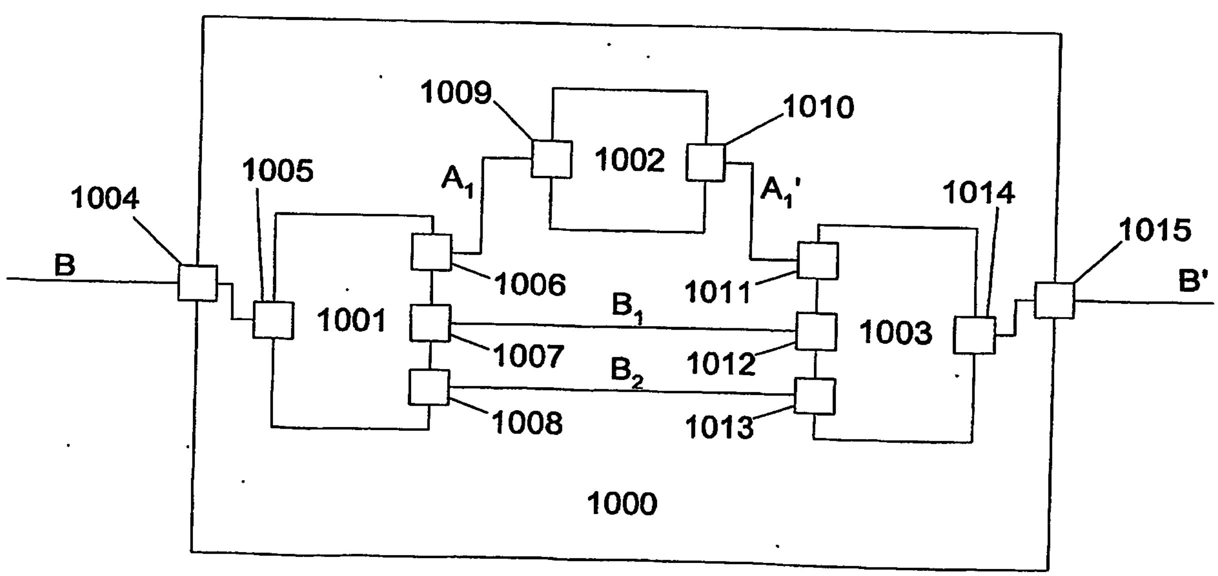
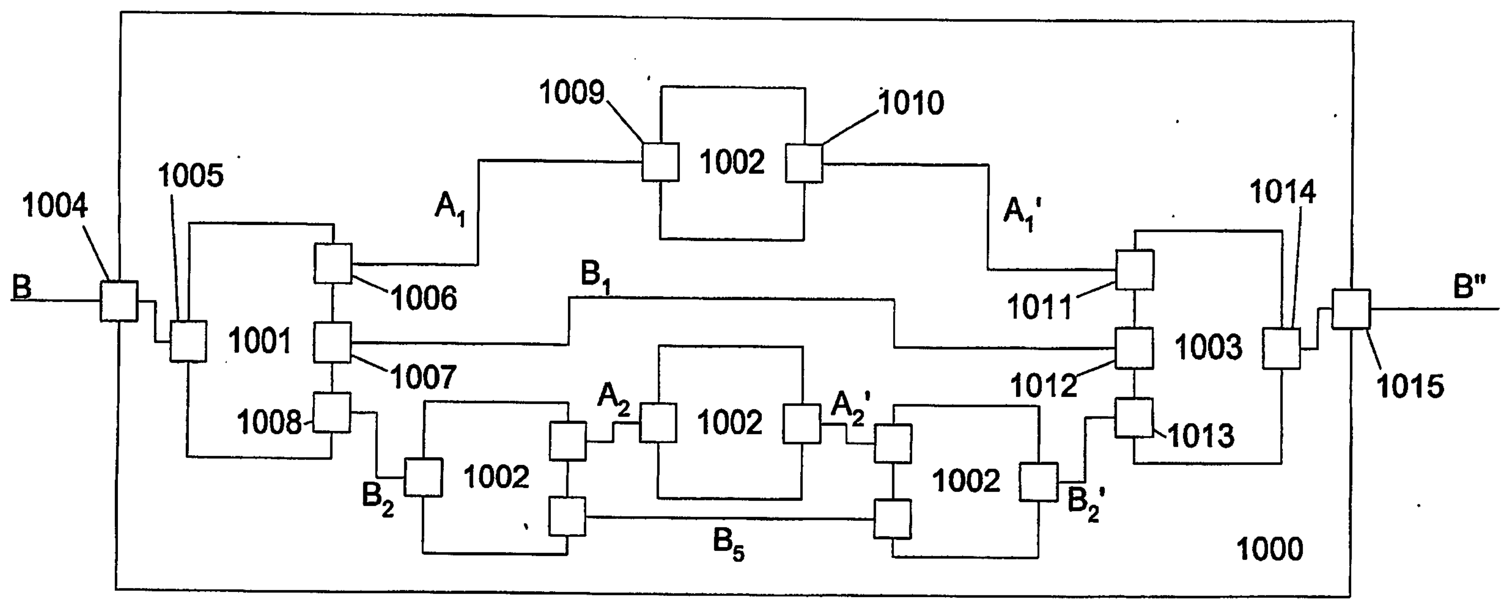

1009
1010
1002
1005
A1
A1'
1014
1004
1015
B
1006
1011
B'
1001
B1
1003
1007
1012
B2
1008
1013
1000

Fig. 40



B
1004
1005
1001
1006
1007
1008
A$_1$
B$_1$
B$_2$
1009
1002
1010
A$_1$'
A$_2$
1002
A$_2$'
1002
1002
B$_5$
B$_2$'
1011
1012
1013
1003
1014
1015
B"
1000


Fig. 41

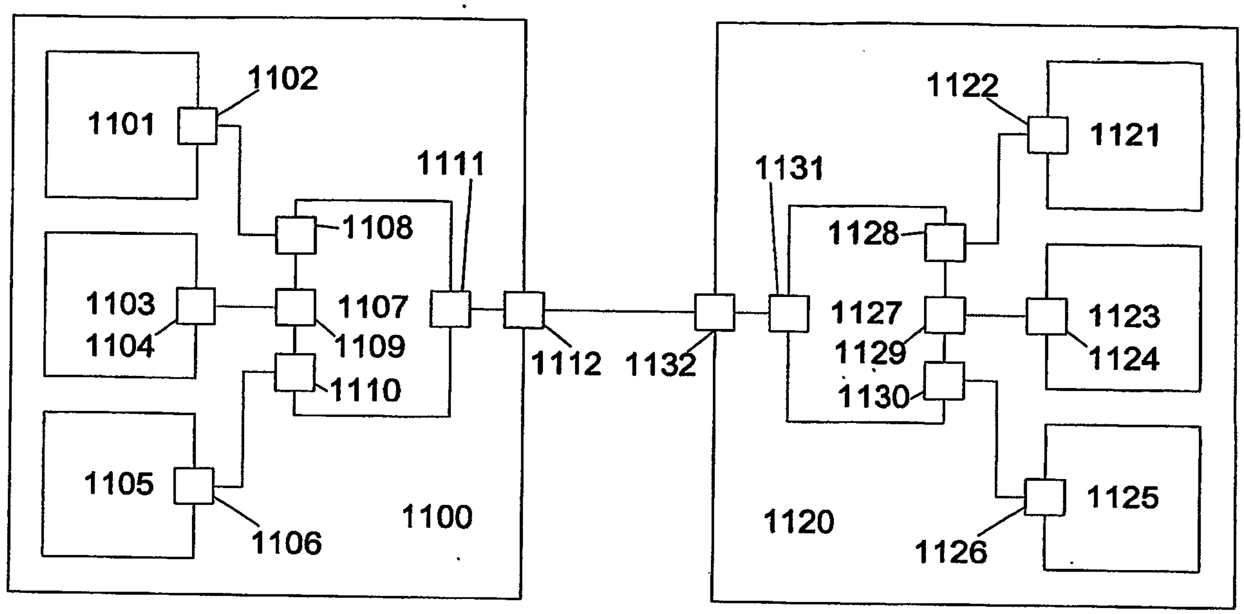



**Fig. 43A**

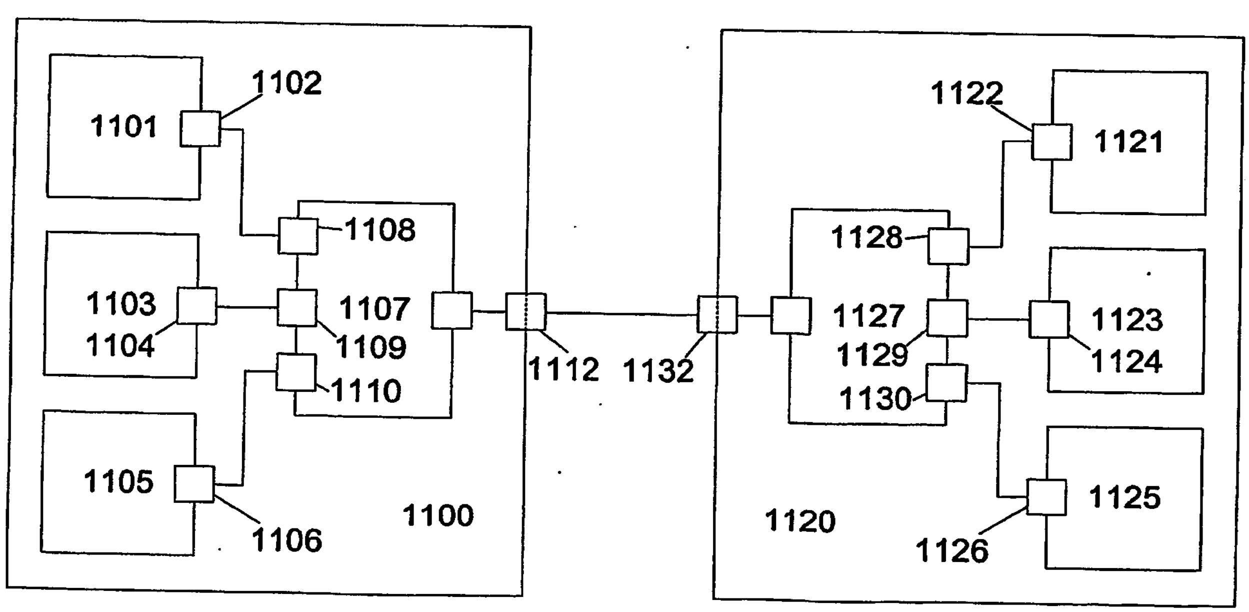



**Fig. 43B**

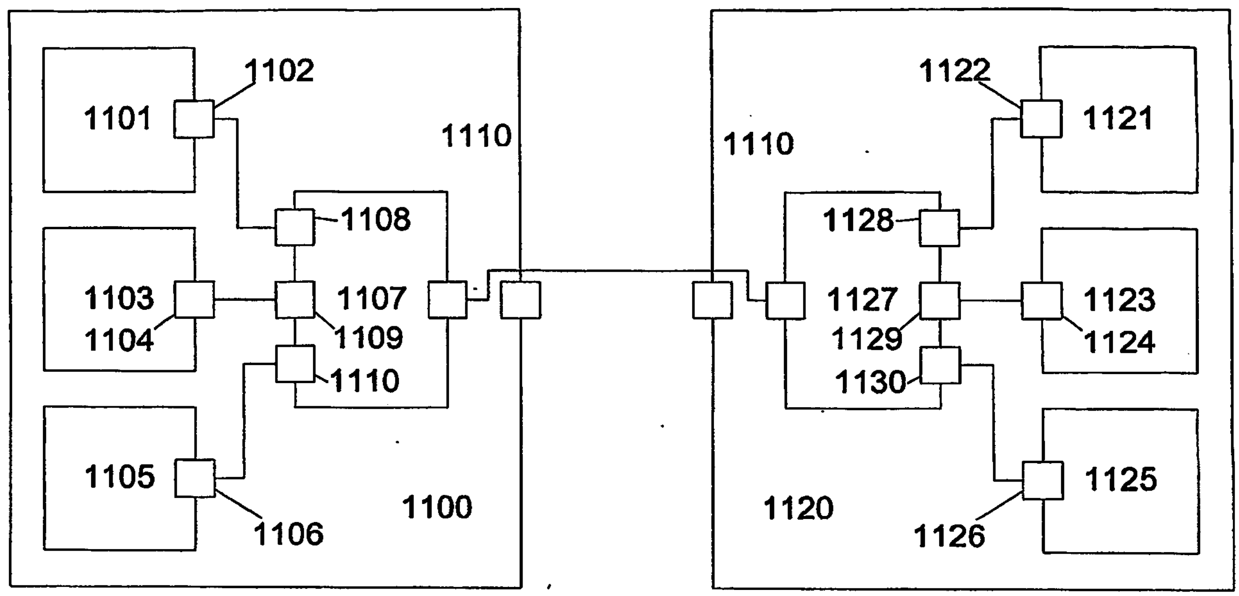



Fig. 43C

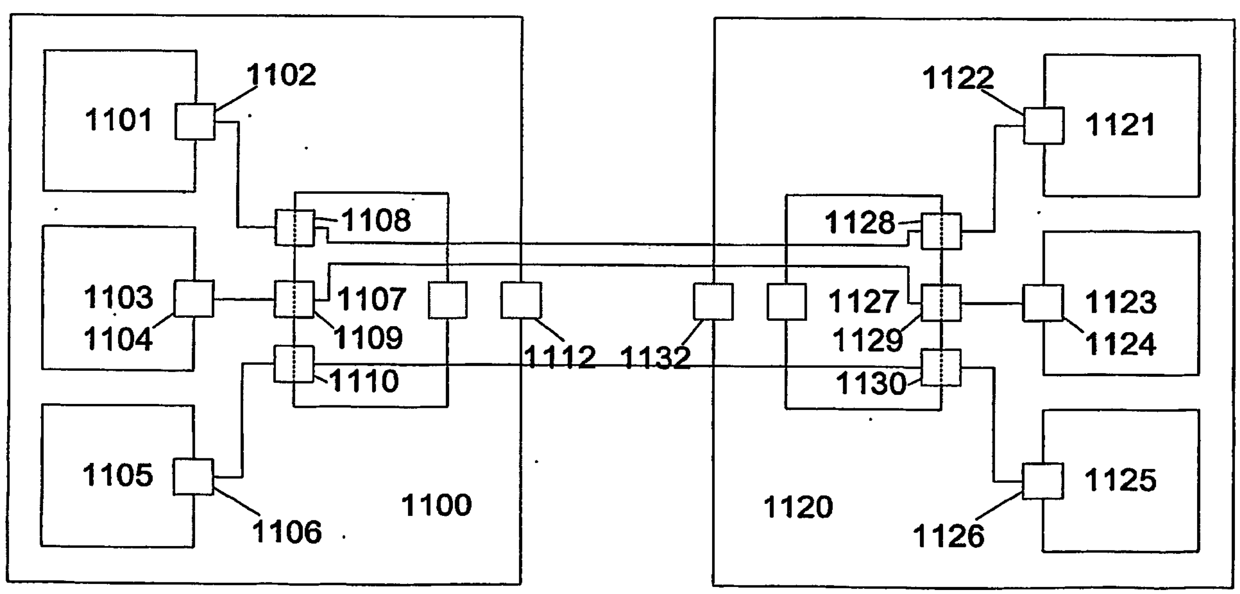



Fig. 43D

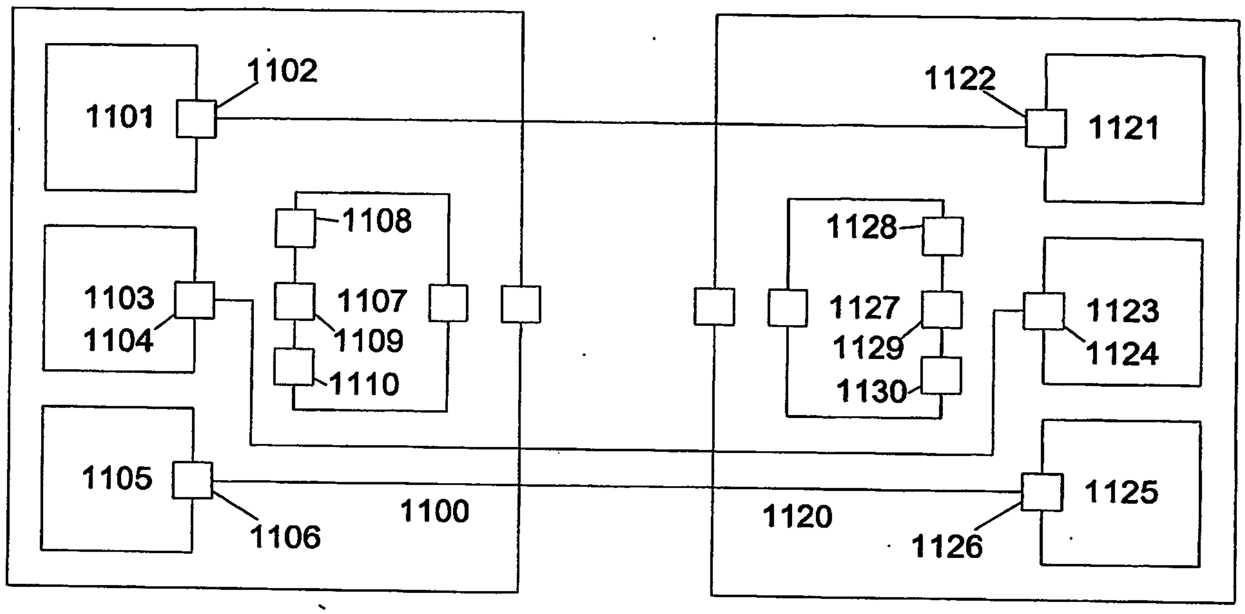



Fig. 43E

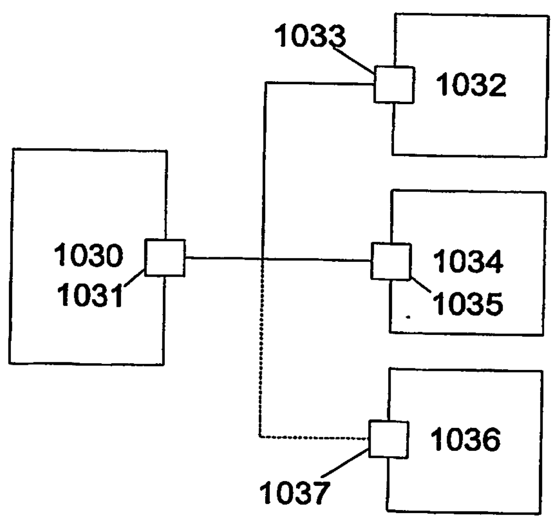



Fig. 42

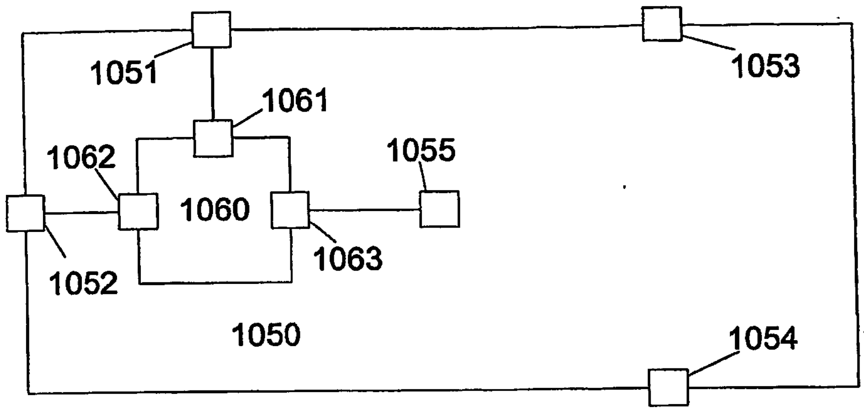



**Fig. 44A**

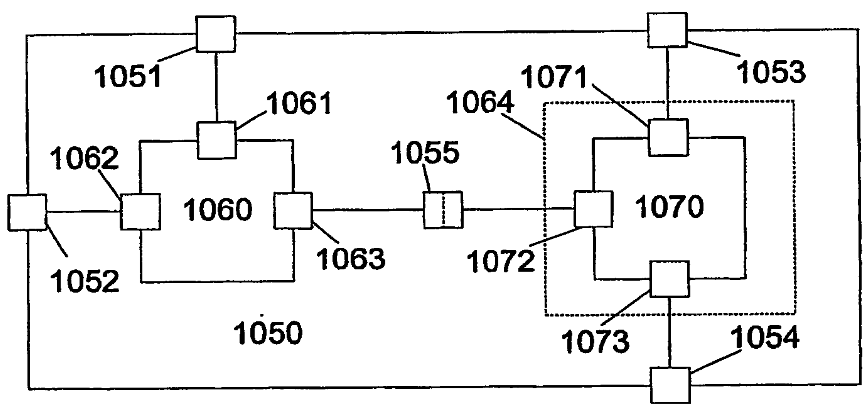



**Fig. 44B**

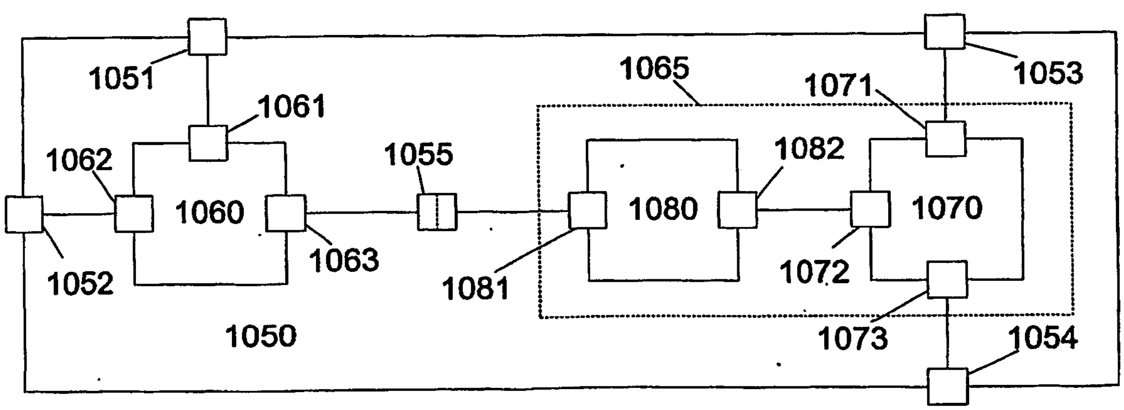



**Fig. 44C**


Fig. 45A


Fig. 45B


Fig. 45C

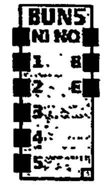



Fig. 45D


Fig. 45E


Fig. 45F


Fig. 45G

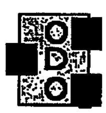



Fig. 45H


Fig. 45I

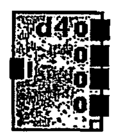



Fig. 45J

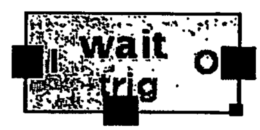



Fig. 45K

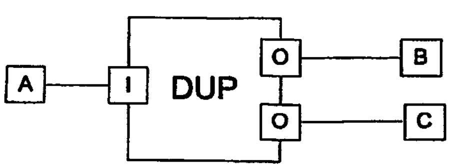



Fig. 45L

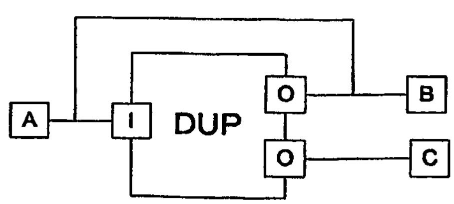



Fig. 45M

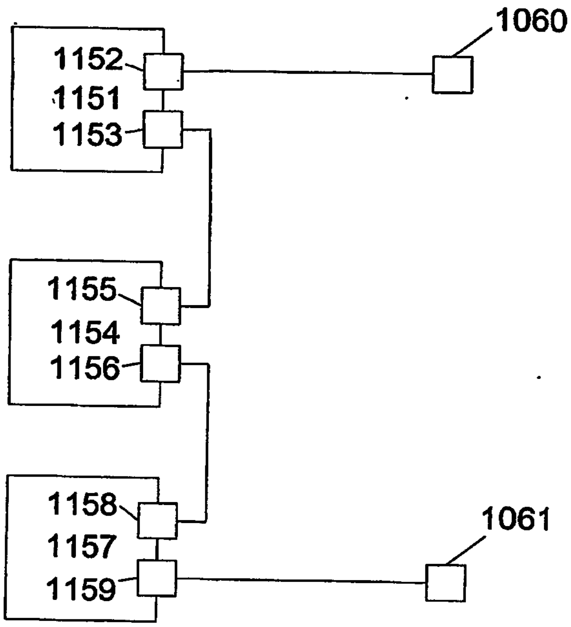



Fig. 48A

| Header |
| --- |
| 1151 |
| 1154 |
| 1157 |

Fig. 48B

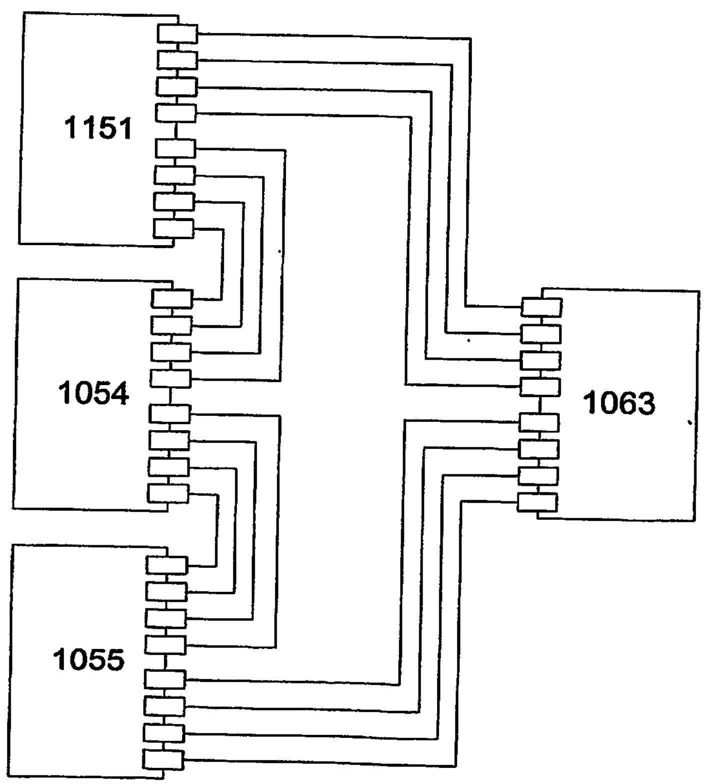



Fig. 48C

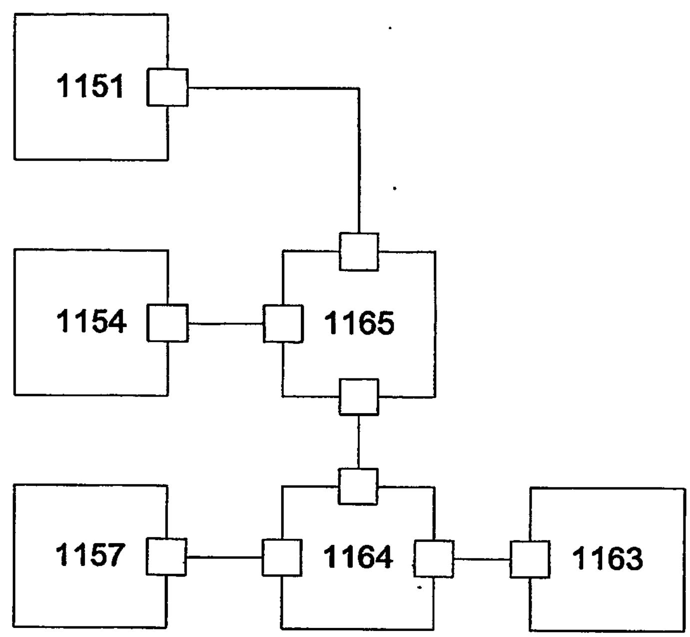



Fig. 48D

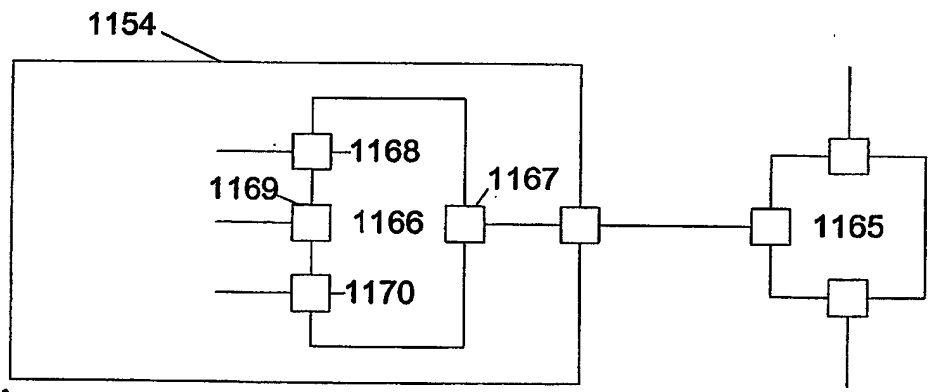



Fig. 48E

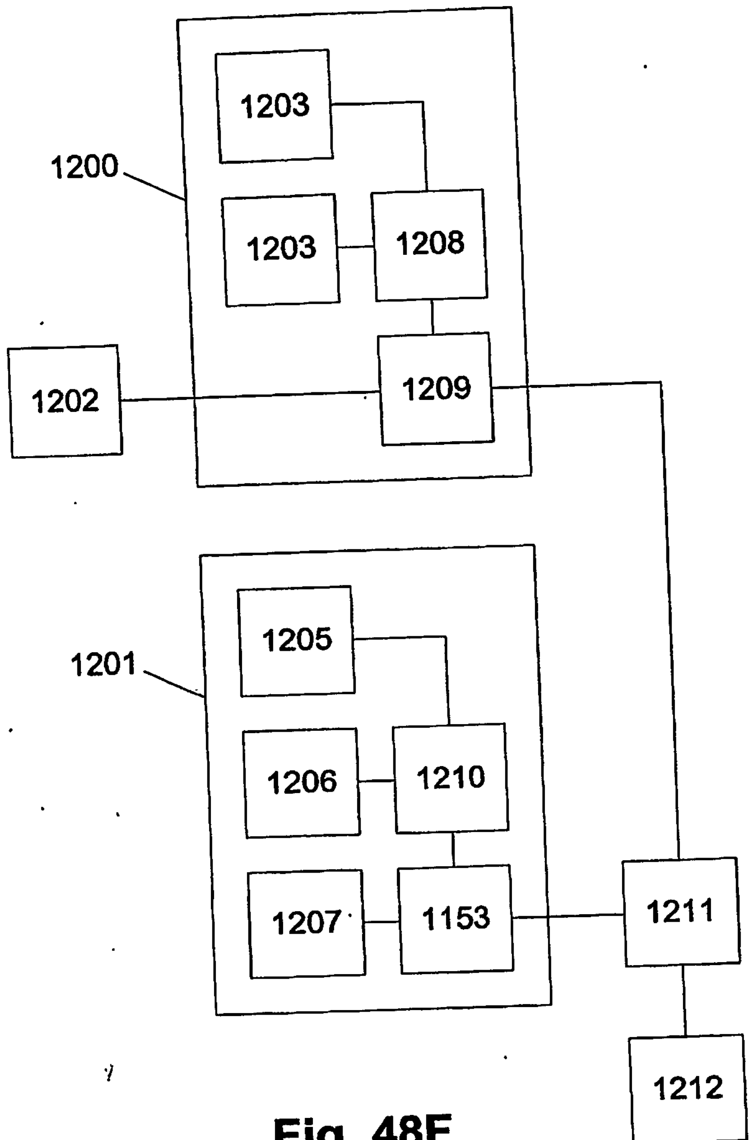



Fig. 48F

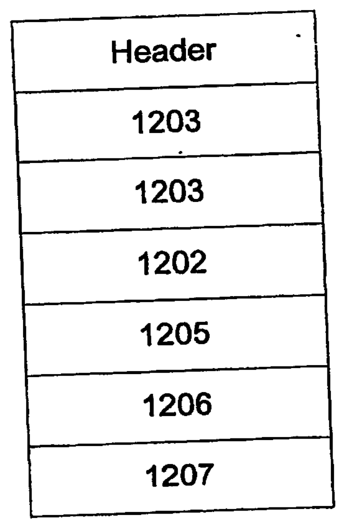



Fig. 48G


Fig. 53A


Fig. 53B

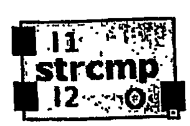



Fig. 53C


Fig. 53D


Fig. 49A

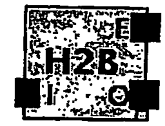



Fig. 49B

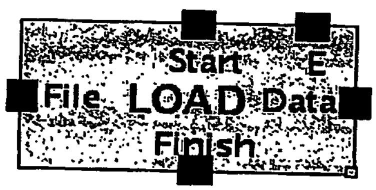



Fig. 50A


Fig. 50B


Fig. 52A

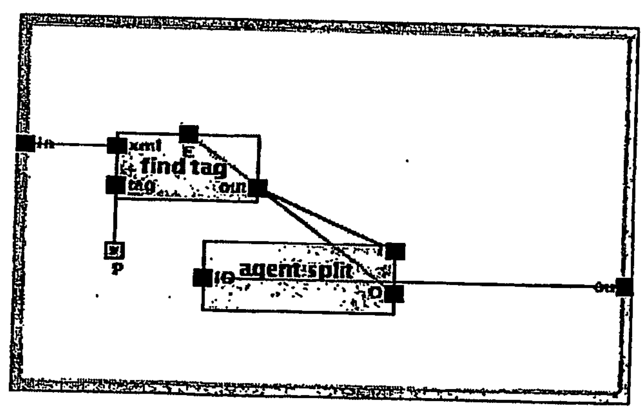



Fig. 52B


Fig. 51A


Fig. 51B


Fig. 51C


Fig. 51D


Fig. 51E


Fig. 51F


Fig. 51G

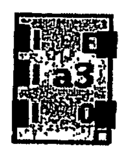



Fig. 51H


Fig. 51I

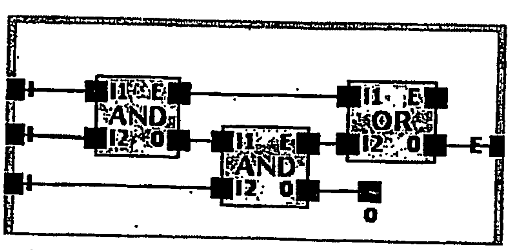



Fig. 51K

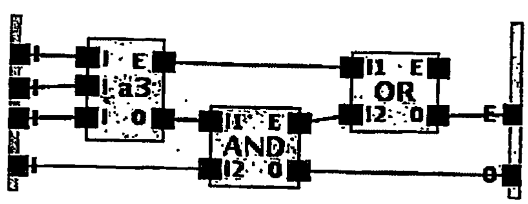



Fig. 51L